Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Операционные системы и системное программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе
на тему

ДИСПЕТЧЕР ПРОЦЕССОВ И ПОТОКОВ

БГУИР КР 1-40 02 01 118 ПЗ

Студент Д.А. Снитко

Руководитель А.О. Игнатович

МИНСК 2024

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет: КСиС. Кафедра: ЭВМ.
Специальность: 40 02 01 «Вычислительные машины, системы и сети».
Специализация: 400201-01 «Проектирование и применение локальных компьютерных сетей».

УТВЕРЖДАЮ
Заведующий кафедрой ЭВМ
_____ Б.В. Никульшин
«____» _____ 2024 г.

ЗАДАНИЕ
по курсовому проекту студента
Снитко Даниила Александровича

1 Тема проекта: «Диспетчер процессов и потоков»
2 Срок сдачи студентом законченного проекта: 10 мая 2024 г.
3 Исходные данные к проекту: нет.
4 Содержание пояснительной записки (перечень подлежащих разработке вопросов):
Титульный лист.
Реферат.
Введение.
1. Обзор литературы.
2. Системное проектирование.
3. Функциональное проектирование.
4. Разработка программных модулей.
5. Руководство пользователя.
Заключение.
Список использованных источников.
Приложения.
5 Перечень графического материала (с точным указанием обязательных чертежей):
5.1 Структурная схема.
5.2 Схема алгоритма `get_thread_info`
5.3 Схема алгоритма `count_cpu_cores`
5.4 Ведомость документов

КАЛЕНДАРНЫЙ ПЛАН

Наименование этапов курсового проекта	Объем этапа, %	Срок выполнения этапа	Примечания
Выбор темы курсового проекта	5	17.02 – 01.03	
Начальный этап ПЗ	30	01.03 – 01.04	
Основная часть кода	50	01.04 – 01.05	
Оформление пояснительной записки и графического материала	15	01.05 – 10.05	с выполнением чертежа
Защита курсового проекта		28.05 – 10.06	

Дата выдачи задания: 20.02.2024 г.

Руководитель А.О. Игнатович

ЗАДАНИЕ ПРИНЯЛ К ИСПОЛНЕНИЮ _____

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 5
1 ОБЗОР ЛИТЕРАТУРЫ 6
1.1 Основные шаги разработки диспетчера процессов и потоков 6
1.2 Постановка задачи 7
1.3 Обзор существующих аналогов 7
1.4 Сравнительный анализ 22
2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ 24
2.1 Блок ввода-вывода 25
2.2 Блок чтения данных 25
2.3 Блок управления процессами и потоками 25
2.4 Блок главного цикла программы 25
3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ 27
3.1 Описание основных структур данных программы 26
3.2 Описание основных функций программы 28
4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ 31
4.1 Разработка структурной схемы 31
4.2 Схемы алгоритмов 31
4.2.1 Схема алгоритма `get_thread_info` 31
4.2.2 Схема алгоритма `count_cpu_cores` 31
4.3 Разработка алгоритмов 31
4.3.1 Алгоритм функции `read_sysinfo` 31
4.3.2 Алгоритм функции `read_processes` 32
4.4 Код программы 33
5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 34
5.1 Требования к программному и аппаратному обеспечению 34
5.2 Руководство по использованию 34
ЗАКЛЮЧЕНИЕ 41
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 42
ПРИЛОЖЕНИЕ А 43
ПРИЛОЖЕНИЕ Б 44
ПРИЛОЖЕНИЕ В 45
ПРИЛОЖЕНИЕ Г 46
ПРИЛОЖЕНИЕ Д 47

ВВЕДЕНИЕ

Диспетчеры процессов и потоков – компоненты, утилиты операционной системы, предназначенные для управления выполнением задач и системным ресурсами, и мониторингом. В Unix-подобных системах, таких как Linux, процессы и потоки играют особую роль.

Процесс в Unix-системе - это запущенная программа или программный код с собственной областью памяти и состоянием. Каждый процесс имеет уникальный идентификатор процесса PID (Process Identifier) и может быть однозначно идентифицирован в системе. Каждый процесс может быть создан другим процессом, называемым родительским процессом. Таким образом, процессы могут образовывать иерархическую структуру.

С другой стороны, потоки - это легкие единицы выполнения внутри процесса. Поток - наборы инструкций, которые выполняются независимо друг от друга в контексте процесса. Потоки в одном процессе имеют общую область памяти и могут обмениваться данными и ресурсами. У них есть свой собственный идентификатор TID (Thread Identifier), который помогает системе управлять потоками. Идентификатор потока (TID) - это целое число, которое операционная система присваивает каждому потоку в процессе. Когда создается поток, операционная система присваивает ему уникальный TID, который остается неизменным на протяжении всего времени существования потока.

Менеджер процессов и потоков, как и утилита Top, предоставляет пользователю информацию о текущих процессах и потоках в системе. Он отображает список запущенных процессов с различными характеристиками, такими как PID, имя пользователя, приоритет, использование ресурсов оперативной памяти и процессора и другие параметры. Эта информация позволяет пользователям отслеживать активность процессов, определять их важность и эффективность использования ресурсов.

Менеджер отображает список процессов, запущенных в системе. Каждый процесс представлен отдельной строкой и содержит информацию о его идентификаторе (PID), пользователе, использовании процессора, памяти и других параметрах. Информация об использовании системных ресурсов, ЦП и памяти.

Целью курсового проекта является разработка диспетчера процессов и потоков для Unix-подобных систем, таких как Linux. Диспетчер должен предоставлять пользователю информацию о текущих процессах и потоках в системе включая PID, имя пользователя, приоритет, использование ресурсов оперативной памяти и процессора и другие параметры. Пользователи должны иметь возможность завершать, сортировать и просматривать активные процессы и потоки. А также для просматривать общую информацию о системе: текущее время, количество пользователей, процессов зомби, среднюю загруженность системы за разное время, процент использования процессора пользователями, процент использования процессора системой и прочую информацию.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Основные шаги разработки диспетчера процессов и потоков

Проектирование структуры данных: Для хранения информации о процессах и потоках необходимо создать соответствующие структуры данных. В нашем случае, это структуры Process и Thread. Они должны содержать необходимую информацию, такую как идентификатор (PID или TID), имя пользователя, приоритет, использование виртуальной и резидентной памяти, использование процессора и команду, запустившую процесс.

Чтение данных о процессах и потоках: Для получения информации о процессах и потоках необходимо прочитать данные из системных файлов. В Unix-подобных системах, таких как Linux, эта информация хранится в файлах в каталоге /proc. Функции read_processes и read_threads отвечают за чтение этих данных и заполнение структур Process и Thread. Для работы с каталогами используются системные вызовы opendir, readdir, closedir, а для работы с файлами - fopen, fgets, sscanf, fclose.

Обработка ввода пользователя: Для взаимодействия с пользователем необходимо реализовать обработку ввода. Функция handle_input отвечает за это. Она обрабатывает нажатия клавиш и выполняет соответствующие действия, такие как переключение между режимами отображения процессов и потоков, сортировка по разным критериям, убийство процессов и потоков и т.д. Для обработки ввода с клавиатуры используются системные вызовы getchar и kbhit.

Отображение информации: Для отображения информации о процессах и потоках необходимо реализовать функции display_processes и display_threads. Они выводят информацию в табличном виде, используя данные из структур Process и Thread. Для ввода-вывода информации в консоль используются системные вызовы printf и scanf, а для форматированного вывода строк - sprintf. Для интерфейса программы будет использоваться библиотека ncurses, которая предоставляет набор функций для создания текстовых пользовательских интерфейсов (TUI) в терминальном окне.

Сортировка: Для сортировки процессов и потоков по разным критериям необходимо реализовать соответствующие функции. В нашем случае, это функции sort_processes_by_ и sort_threads_by_. Они используют стандартную функцию qsort для сортировки массивов структур. Для сравнения критериев сортировки, введенных пользователем, используется системный вызов strcmp.

Завершение процессов и потоков: Для завершения процессов и потоков необходимо реализовать функции kill_process_by_pid и kill_thread_by_tid. Они используют системные вызовы kill и pthread_cancel для отправки сигнала завершения процессу или потоку.

Главный цикл программы: Главный цикл программы выполняет следующие действия: очистка экрана, чтение данных о процессах и потоках, отображение информации, ожидание 1 секунда и обработка ввода пользователя. Этот цикл повторяется бесконечно, пока программа не будет завершена. Для

выполнения системных команд используется системный вызов system, а для приостановки выполнения программы на указанное количество секунд - sleep.

Обработка сигналов: Для корректного завершения программы необходимо обрабатывать сигнал SIGINT, который генерируется при нажатии клавиш Ctrl+C. Функция handle_signal отвечает за это. Она выводит сообщение о завершении программы и вызывает функцию exit для завершения программы. Для установки обработчика сигнала используется системный вызов signal.

1.2 Постановка задачи

В рамках данного проекта будет разработан диспетчер процессов и потоков, обладающий функциональностью мониторинга процессов и управлением, запущенных в данный момент. Диспетчер должен предоставлять информацию о PID (идентификатор процесса), пользователе, приоритете, потреблении виртуальной и физической памяти, % времени процессора, % ОЗУ используемым процессором, название команды, инициализировавшей процесс. Для реализации данного диспетчера будет использован язык программирования высокого уровня, такой как Си. В качестве операционной системы была выбрана Fedora Workstation 39.

1.3 Обзор существующих аналогов

Существует множество программ для мониторинга и управления процессами в операционных системах. Два из наиболее известных и широко используемых аналогов программы диспетчера процессов и потоков, разработанной в рамках данного проекта, это утилита top и программа htop.

1.3.1 top

Top (table of processes) является стандартной утилитой Unix-подобных операционных систем для мониторинга процессов. Она предоставляет пользователю динамическое представление о текущем состоянии процессов в системе, включая их идентификаторы, использование ресурсов, приоритет, статус и т.д.

Программа top периодически обновляет информацию о процессах, сортирует ее в соответствии с выбранным критерием (по умолчанию - использование процессора) и выводит ее на экран. Пользователь может взаимодействовать с программой, используя различные команды для сортировки, фильтрации и управления процессами. Например, можно отправить сигнал для завершения процесса или изменить приоритет процесса.

Top широко используется системными администраторами и разработчиками для мониторинга и диагностики производительности системы. Однако, несмотря на ее популярность, у программы есть некоторые недостатки,

такие как отсутствие графического интерфейса и неудобство использования для новичков.

```
top - 16:54:41 up  5:12,  2 users,  load average: 0.75, 0.77, 0.35
Tasks: 116 total,   1 running, 115 sleeping,   0 stopped,   0 zombie
Cpu(s):  5.3%us,  2.7%sy,  0.0%ni, 91.7%id,  0.3%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    775540k total,   758548k used,    16992k free,    13920k buffers
Swap:   787144k total,    34724k used,   752420k free,   443552k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6938 funalien  15   0 70012  29m  18m S  4.0  3.9  10:40.43 ktorrent
 5375 root      15   0 79060  55m 6616 S  2.3  7.3   4:45.84 Xorg
 7869 funalien  15   0 30400  15m  13m S  1.0  2.0   0:00.99 ksnapshot
 5600 funalien  18   0 15252 9700 4528 S  0.3  1.3   0:19.03 pypanel
 5605 funalien  15   0  9704 3592 2968 S  0.3  0.5   1:20.99 conky
 7802 funalien  15   0  228m  75m  23m S  0.3  9.9   0:36.56 firefox-bin
    1 root      15   0  2952 1852  532 S  0.0  0.2   0:01.33 init
    2 root      11  -5     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/0
    4 root      34  19     0    0    0 S  0.0  0.0   0:00.10 ksoftirqd/0
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    6 root      10  -5     0    0    0 S  0.0  0.0   0:00.16 events/0
    7 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 khelper
   26 root      12  -5     0    0    0 S  0.0  0.0   0:00.00 kblockd/0
   27 root      20  -5     0    0    0 S  0.0  0.0   0:00.00 kacpid
   28 root      20  -5     0    0    0 S  0.0  0.0   0:00.00 kacpi_notify
  108 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 kseriod
```

Рисунок 1.1 – Интерфейс утилиты top

1.3.2 htop

Htop - это усовершенствованная версия программы top, которая предоставляет более удобный и функциональный интерфейс для мониторинга процессов. В отличие от top, htop использует графический интерфейс, позволяющий пользователю просматривать список процессов в виде таблицы, сортировать их по различным критериям, фильтровать по имени или идентификатору, а также управлять ими с помощью мыши или клавиатуры.

Программа htop предоставляет более подробную информацию о процессах, включая использование ресурсов в реальном времени, графики использования процессора и памяти, а также информацию о загрузке системы. Кроме того, htop позволяет пользователю отправлять сигналы процессам, менять их приоритет, завершать или замораживать процессы, а также выполнять другие операции управления.

Htop является более удобным и функциональным инструментом для мониторинга процессов, чем top, и широко используется системными администраторами и разработчиками. Однако, несмотря на свои преимущества, htop также имеет некоторые недостатки, такие как более высокие требования к ресурсам системы и невозможность работы в текстовом режиме.

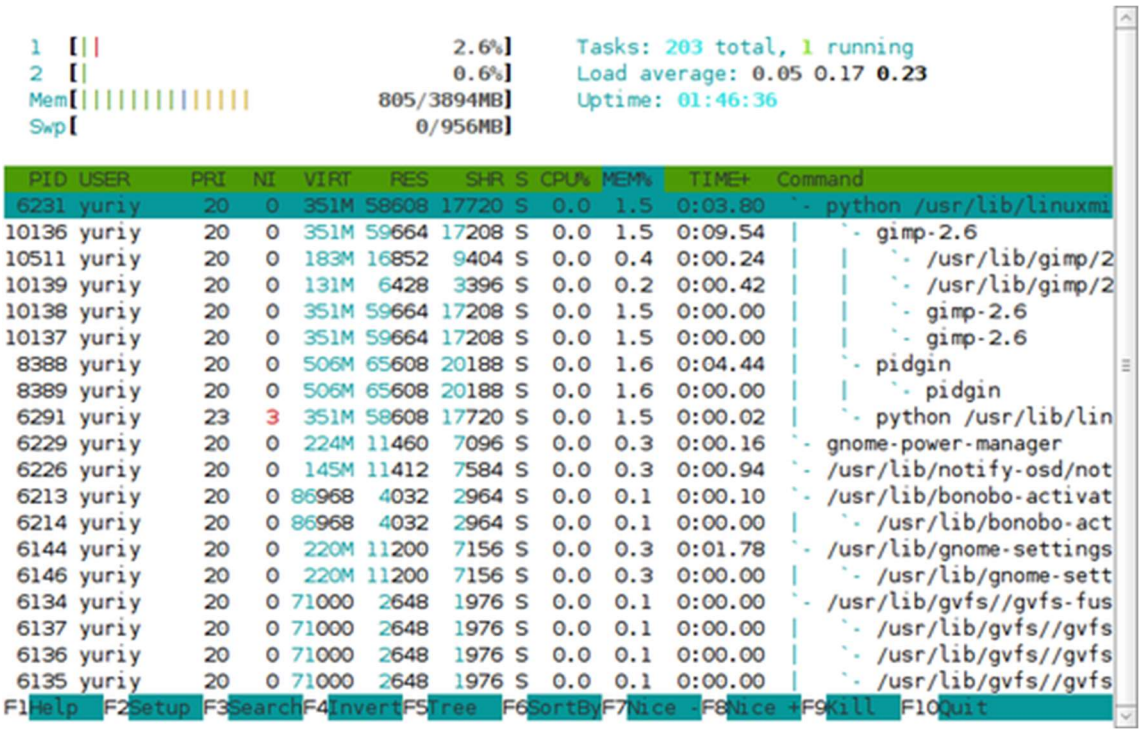


Рисунок 1.2 – Интерфейс утилиты htop

1.3.3 atop

Atop имеет два режима работы — сбор статистики и наблюдение за системой в реальном времени. В режиме сбора статистики atop запускается как демон и раз в N времени (обычно 10 мин) скидывает состояние в двоичный журнал. Потом по этому журналу atop'ом же (ключ -r и имя лог-файла) можно бегать вперёд-назад кнопками T и t, наблюдая показания atop'а с усреднением за 10 минут в любой интересный момент времени.

В отличие от top отлично знает про существование блочных устройств и сетевых интерфейса, способен показывать их загрузку в процентах (на 10G, правда, процентов не получается, но хотя бы показывается количество мегабит).

Незаменимое средство для поиска источников лагов на сервере, так как сохраняет не только статистику загрузки системы, но и показатели каждого процесса — то есть «долистав» до нужного момента времени можно увидеть, кто этот счастливый момент с LA > 30 создал. И что именно было причиной — IO программ, своп (нехватка памяти), процесор или что-то ещё. Помимо большего количества информации ещё способен двумя цветами подсказывать, какие параметры выходят за разумные пределы.

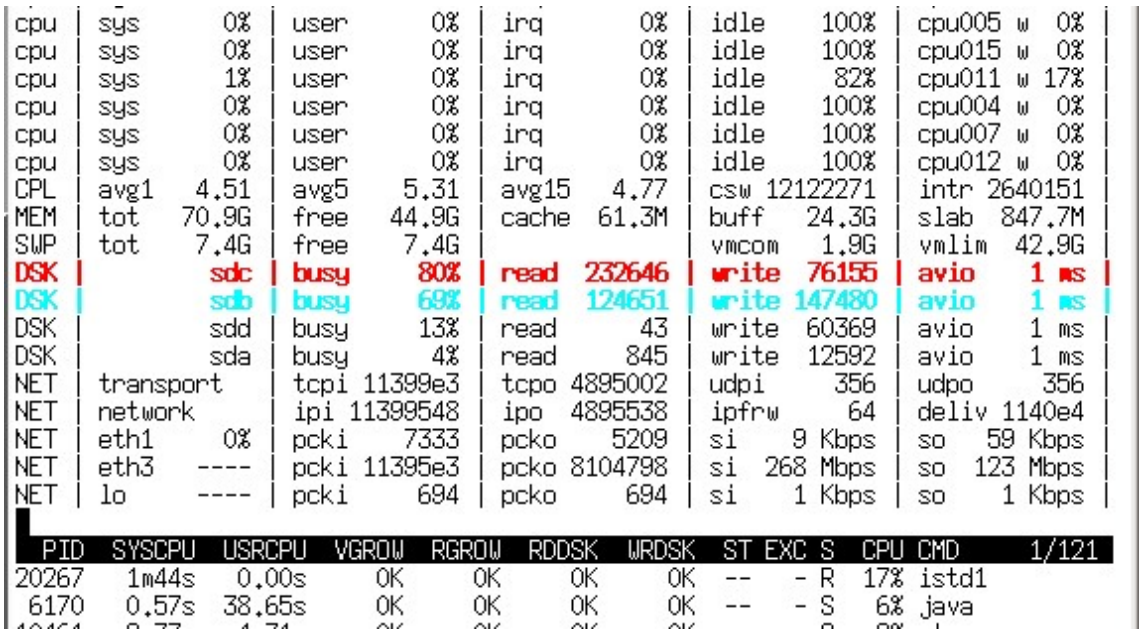


Рисунок 1.3 – Интерфейс утилиты atop

1.3.4 iotop

Iotop является утилитой для мониторинга дисковой активности в системах Linux. Эта утилита показывает, какие процессы в настоящее время выполняют ввод-вывод с диском, сколько байт они читают или записывают, а также другие полезные сведения.

Iotop похож на утилиту top, которая используется для мониторинга использования процессора и памяти, но вместо этого концентрируется на дисковой активности. Это может быть полезно при диагностике проблем с производительностью, вызванных избыточной нагрузкой на диск, или при идентификации процессов, которые выполняют слишком много операций ввода-вывода.

Iotop может работать в двух режимах: батчевом и интерактивном. В батчевом режиме iotop выводит информацию о дисковой активности один раз и завершает работу, тогда как в интерактивном режиме он обновляет информацию в реальном времени.

Некоторые из ключевых параметров, которые можно использовать с iotop, включают:

`-o`: отсортировать вывод по указанному столбцу.

`-p`: мониторить только указанные процессы.

`-q`: запустить iotop в бесшумном режиме, без вывода статистики по умолчанию.

`-t`: отображать время простоя для каждого процесса.

Iotop является очень полезным инструментом для администрирования систем Linux, и его использование может помочь выявить и устранить проблемы с производительностью, связанные с дисковой активностью.

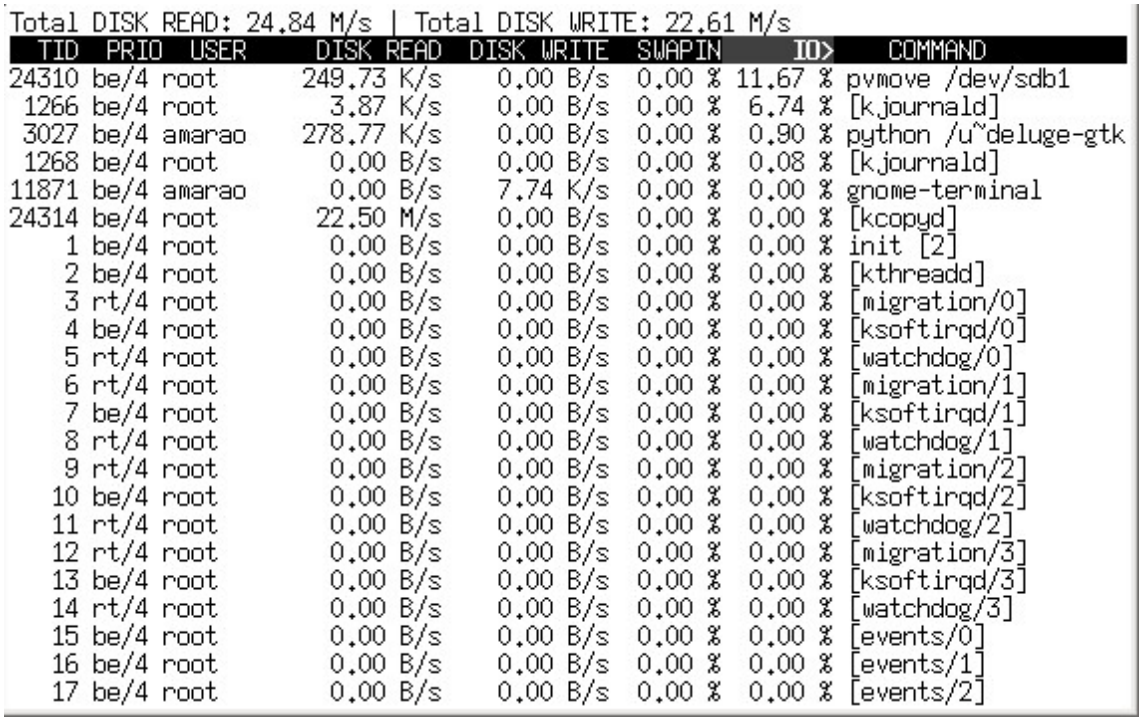


Рисунок 1.4 – Интерфейс утилиты iotop

1.3.5 iftop

Iftop - это утилита командной строки для мониторинга трафика сети в реальном времени. Эта программа отображает количество передаваемых и принимаемых байтов для каждого сетевого соединения и позволяет отслеживать использование пропускной способности сети.

Iftop работает путем анализа пакетов, проходящих через сетевой интерфейс, и отображает информацию о них в удобном для чтения формате. По умолчанию, iftop сортирует соединения по скорости передачи данных, но также можно использовать различные параметры для сортировки по другим критериям, таким как использование пропускной способности, количество пакетов и т.д.

Некоторые из ключевых параметров, которые можно использовать с iftop, включают:

`-i`: выбрать сетевой интерфейс для мониторинга.

`-f`: использовать фильтр пакетов для отображения только определенных соединений.

`-n`: отображать IP-адреса вместо имен хостов.

`-p`: указать порт для мониторинга.

`-b`: отображать скорость передачи данных в битах в секунду вместо байт.

Iftop является очень полезным инструментом для диагностики проблем с сетью, таких как перегрузка пропускной способности, несанкционированное использование сети и т.д.

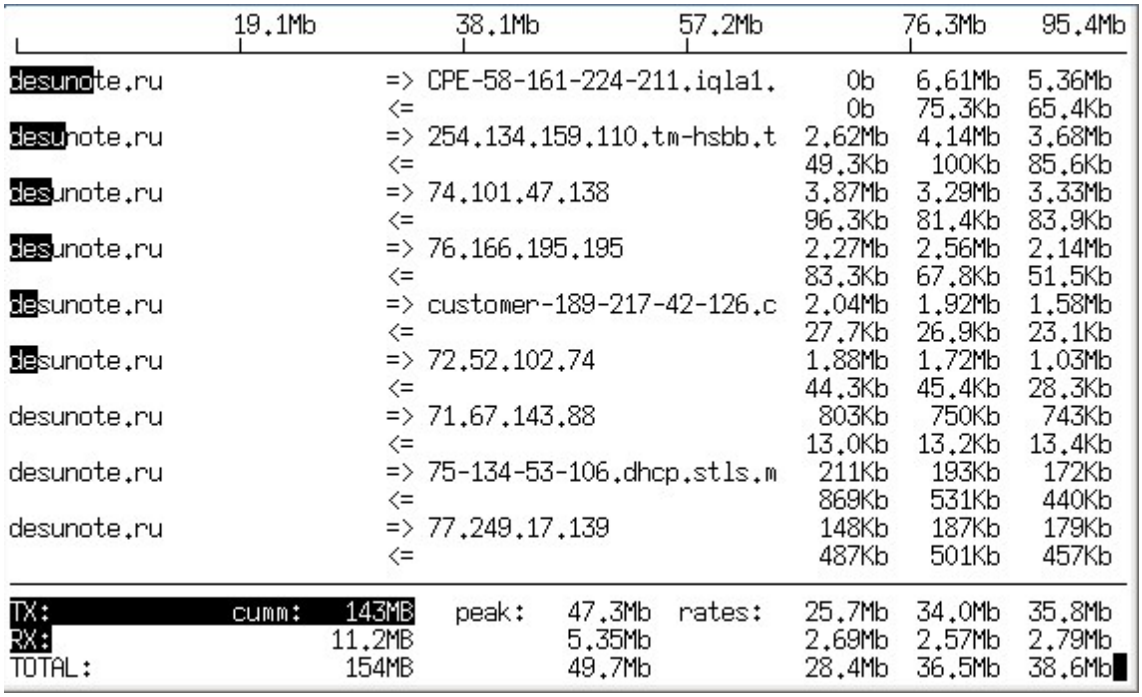


Рисунок 1.5 – Интерфейс утилиты iftop

1.3.6 powertop

Powertop - это утилита для мониторинга и оптимизации энергопотребления ноутбуков и других мобильных устройств с операционной системой Linux. Эта программа помогает выявить процессы и устройства, которые наиболее интенсивно используют энергию, и предлагает рекомендации по их оптимизации.

Powertop работает путем анализа активности процессов и устройств, а также измерения энергопотребления системы. Затем программа отображает список процессов и устройств, отсортированный по уровню энергопотребления, и предоставляет рекомендации по их оптимизации. Например, powertop может предложить снизить яркость экрана, отключить неиспользуемые устройства или изменить параметры режима энергосбережения для определенных процессов.

Некоторые из ключевых параметров, которые можно использовать с powertop, включают:

`-c`: показать список процессов и устройств в двух столбцах для удобства просмотра.

`-d`: запустить powertop в режиме мониторинга, без рекомендаций по оптимизации.

`-t`: отображать только процессы, превышающие заданный порог энергопотребления.

`-w`: запустить powertop в режиме калибровки, для более точного измерения энергопотребления.

Powertop является очень полезным инструментом для оптимизации энергопотребления ноутбуков и других мобильных устройств с операционной системой Linux. Эта утилита доступна для большинства дистрибутивов Linux и может быть установлена с помощью стандартного менеджера пакетов.

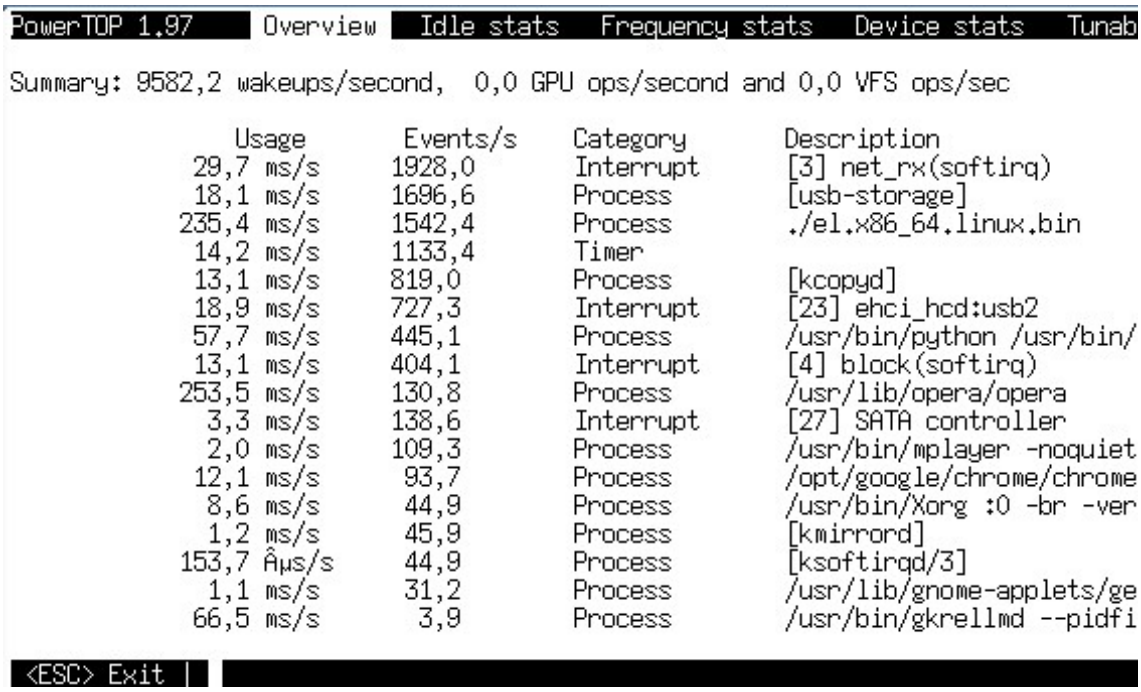


Рисунок 1.6 – Интерфейс утилиты powertop

1.3.7 itop

Itop - это утилита командной строки для мониторинга использования ресурсов системы в операционной системе Linux. Эта программа отображает список запущенных процессов и их использование ресурсов, таких как ЦП, память, диск и сеть, в реальном времени.

Itop работает путем анализа информации о процессах из /proc и отображает ее в удобном для чтения формате. По умолчанию, itop сортирует процессы по использованию ЦП, но также можно использовать различные параметры для сортировки по другим критериям, таким как использование памяти, диска или сети.

Некоторые из ключевых параметров, которые можно использовать с itop, включают:

-o: отсортировать процессы по указанному столбцу.

-p: мониторить только указанные процессы.

-u: отображать только процессы, запущенные от имени указанного пользователя.

-d: указать интервал обновления списка процессов в секундах.

-a: отображать все процессы, включая те, которые не используют ресурсы системы.

Itop является очень полезным инструментом для диагностики проблем с производительностью системы и оптимизации использования ресурсов. Эта утилита доступна для большинства дистрибутивов Linux и может быть установлена с помощью стандартного менеджера пакетов.

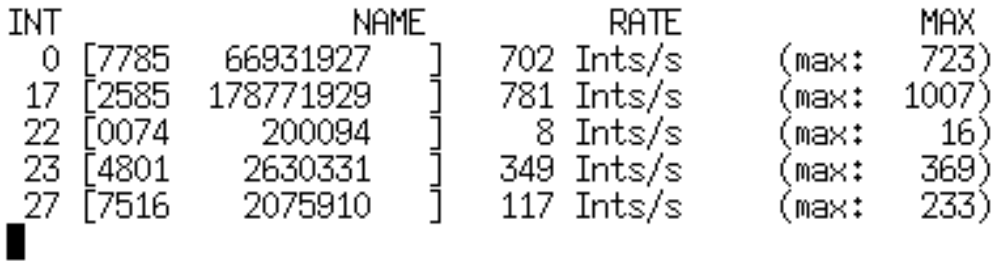


Рисунок 1.7 – Интерфейс утилиты itop

1.3.8 dnstop

Dnstop - это утилита командной строки для мониторинга DNS-трафика в операционной системе Linux. Эта программа отображает статистику по DNS-запросам и ответам, а также позволяет отслеживать активность конкретных хостов и доменов в реальном времени.

Dnstop работает путем анализа пакетов, проходящих через сетевой интерфейс, и извлечения из них информации о DNS-запросах и ответах. Затем программа отображает статистику по количеству запросов и ответов, типам запросов, кодам ответов, используемым протоколам и другим параметрам. Кроме того, dnstop позволяет отслеживать активность конкретных хостов и доменов, отображая список наиболее активных из них и их статистику.

Некоторые из ключевых параметров, которые можно использовать с dnstop, включают:

-i: указать сетевой интерфейс для мониторинга.

-f: использовать фильтр пакетов для отображения только определенных DNS-запросов и ответов.

-n: отображать IP-адреса вместо имен хостов.

-p: указать порт для мониторинга (по умолчанию используется порт 53).

-l: записывать статистику в файл для последующего анализа.

Dnstop является очень полезным инструментом для диагностики проблем с DNS-трафиком, таких как замедление работы сети, несанкционированное использование DNS-сервера и т.д. Эта утилита доступна для большинства дистрибутивов Linux и может быть установлена с помощью стандартного менеджера пакетов.

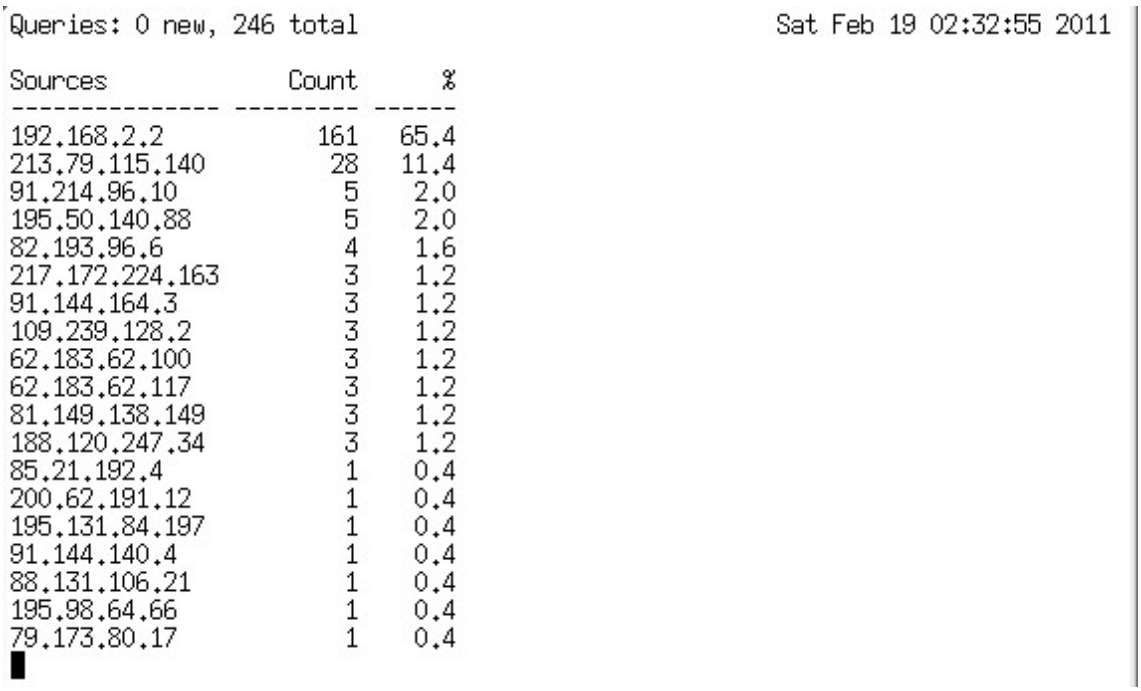


Рисунок 1.8 – Интерфейс утилиты dnstop

1.3.9 jnettop

Jnettop - это утилита командной строки для мониторинга сетевой активности Java-приложений в операционной системе Linux. Эта программа отображает список активных сетевых соединений, используемых Java-приложениями, и их статистику, такую как скорость передачи данных, объем переданных данных и время активности.

Jnettop работает путем анализа информации о сетевых соединениях, предоставляемой виртуальной машиной Java (JVM), и отображает ее в удобном для чтения формате. По умолчанию, jnettop сортирует соединения по скорости передачи данных, но также можно использовать различные параметры для сортировки по другим критериям, таким как объем переданных данных или время активности.

Некоторые из ключевых параметров, которые можно использовать с jnettop, включают:

-l: отображать информацию о локальных соединениях (только для соединений, установленных на локальном хосте).

-r: отображать информацию о удаленных соединениях (только для соединений, установленных с удаленных хостов).

-p: фильтровать список соединений по указанному порту.

-i: указать интервал обновления списка соединений в секундах.

-m: отображать статистику по использованию памяти для каждого Java-приложения.

Jnettop является очень полезным инструментом для диагностики проблем с сетевой активностью Java-приложений, таких как замедление работы приложения, несанкционированное использование сети и т.д. Эта утилита доступна для большинства дистрибутивов Linux и может быть установлена с помощью стандартного менеджера пакетов. Однако, для ее работы может потребоваться установка и настройка соответствующего агента мониторинга Java-приложений.

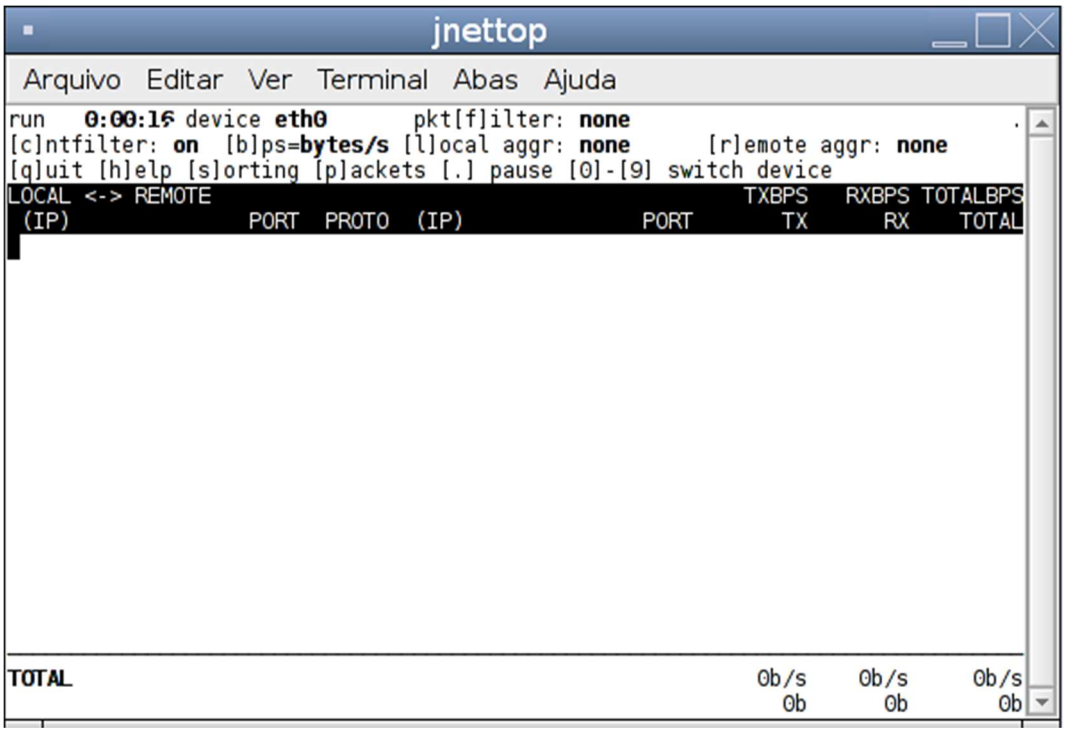


Рисунок 1.9 – Интерфейс утилиты jnettop

1.3.10 xrestop

Xrestop - это графическая утилита для мониторинга использования ресурсов X-сервера в операционной системе Linux. Эта программа отображает список запущенных клиентских приложений и их использование ресурсов, таких как ЦП, память и сеть, в реальном времени.

Xrestop работает путем анализа информации о клиентских приложениях, предоставляемой X-сервером, и отображает ее в удобном для чтения формате. По умолчанию, xrestop сортирует приложения по использованию ЦП, но также можно использовать различные параметры для сортировки по другим критериям, таким как использование памяти или сети.

Некоторые из ключевых параметров, которые можно использовать с xrestop, включают:

`-display`: указать адрес и номер дисплея X-сервера для мониторинга.

`-update`: указать интервал обновления списка приложений в секундах.

-geometry: указать геометрию окна программы (ширину и высоту в пикселях).

-font: указать шрифт для отображения текста в программе.

-help: отобразить справку по использованию программы.

Xrestop является очень полезным инструментом для диагностики проблем с производительностью X-сервера и оптимизации использования ресурсов клиентскими приложениями. Эта утилита доступна для большинства дистрибутивов Linux и может быть установлена с помощью стандартного менеджера пакетов. Однако, для ее работы может потребоваться настройка доступа к X-серверу с использованием механизма аутентификации xauth.

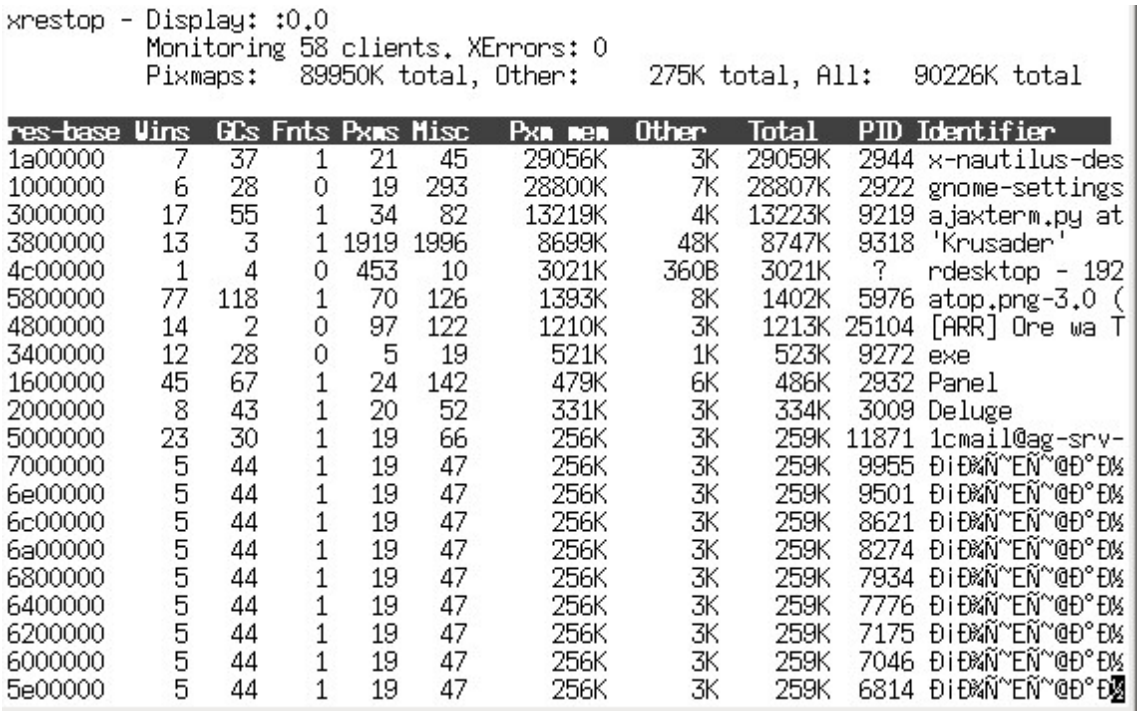


Рисунок 1.10 – Интерфейс утилиты xrestop

1.3.11 slabtop

Slabtop - это утилита командной строки для мониторинга использования кэша ядра (слабов) в операционной системе Linux. Эта программа отображает список кэшей, используемых ядром для хранения часто используемых объектов, и их статистику, такую как количество используемых и свободных объектов, объем памяти, занимаемый кэшами, и т.д.

Slabtop работает путем анализа информации о кэшах, предоставляемой ядром, и отображает ее в удобном для чтения формате. По умолчанию, slabtop сортирует кэши по объему используемой памяти, но также можно использовать различные параметры для сортировки по другим критериям, таким как количество используемых объектов или фрагментация памяти.

Некоторые из ключевых параметров, которые можно использовать с slabtop, включают:

`-o`: отсортировать кэши по указанному столбцу.

`-s`: отсортировать кэши по указанному критерию (например, по объему памяти или количеству объектов).

`-d`: указать интервал обновления списка кэшей в секундах.

`-c`: отображать информацию о фрагментации памяти для каждого кэша.

`-h`: отобразить справку по использованию программы.

Slabtop является очень полезным инструментом для диагностики проблем с использованием памяти в системе, вызванных фрагментацией или нехваткой памяти в кэшах ядра. Эта утилита доступна для большинства дистрибутивов Linux и может быть установлена с помощью стандартного менеджера пакетов. Однако, для ее работы может потребоваться настройка ядра с включением поддержки slab-дебаггера (SLUB debugging).

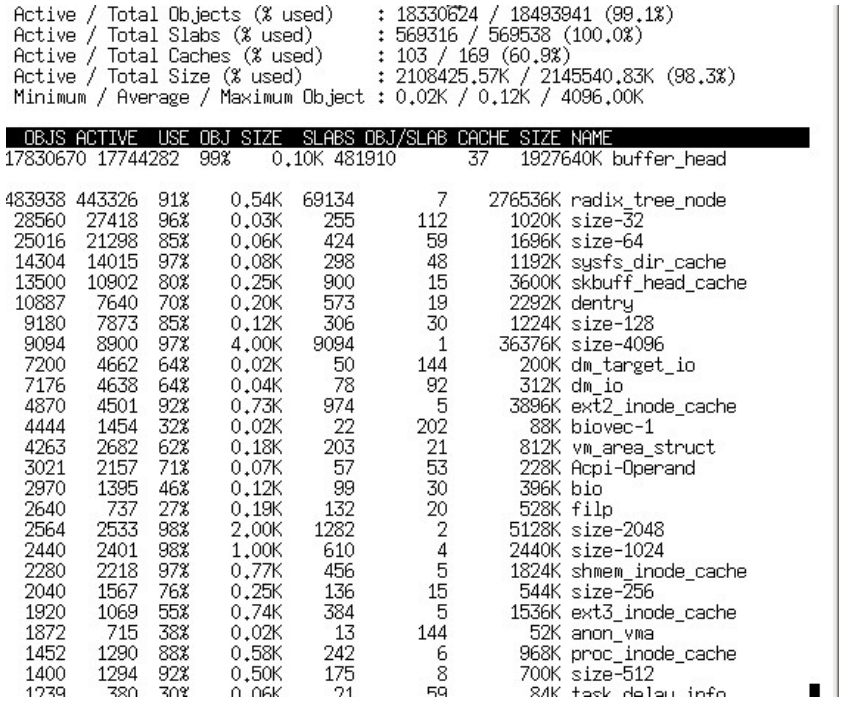


Рисунок 1.11 – Интерфейс утилиты slabtop

1.3.12 mytop

Mytop - это утилита командной строки для мониторинга использования ресурсов базы данных MySQL в операционной системе Linux. Эта программа отображает список запущенных потоков MySQL и их использование ресурсов, таких как ЦП, память и диск, в реальном времени.

Mytop работает путем анализа информации о потоках, предоставляемой MySQL, и отображает ее в удобном для чтения формате. По умолчанию, mytop сортирует потоки по использованию ЦП, но также можно использовать

различные параметры для сортировки по другим критериям, таким как время выполнения запроса или использование памяти.

Некоторые из ключевых параметров, которые можно использовать с mytop, включают:

`-h`: указать адрес хоста, на котором запущена база данных MySQL.

`-u`: указать имя пользователя для подключения к базе данных MySQL.

`-p`: указать пароль для подключения к базе данных MySQL.

`-d`: указать имя базы данных для мониторинга (по умолчанию используется текущая база данных).

`-i`: указать интервал обновления списка потоков в секундах.

`-s`: отсортировать потоки по указанному критерию (например, по времени выполнения запроса или использованию памяти).

Mytop является очень полезным инструментом для диагностики проблем с производительностью базы данных MySQL и оптимизации использования ресурсов запросами. Эта утилита доступна для большинства дистрибутивов Linux и может быть установлена с помощью стандартного менеджера пакетов. Однако, для ее работы может потребоваться настройка доступа к базе данных MySQL с использованием соответствующих прав доступа.

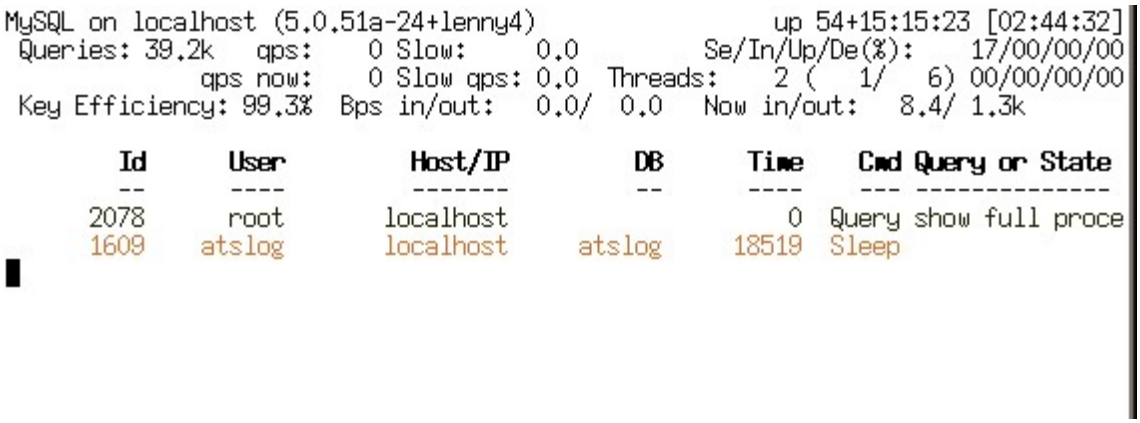


Рисунок 1.12 – Интерфейс утилиты mytop

1.3.13 xentop

Xentop - это утилита командной строки для мониторинга использования ресурсов виртуальных машин (ВМ) в среде Xen. Эта программа отображает список запущенных ВМ и их использование ресурсов, таких как ЦП, память, диск и сеть, в реальном времени.

Xentop работает путем анализа информации о ВМ, предоставляемой гипервизором Xen, и отображает ее в удобном для чтения формате. По умолчанию, xentop сортирует ВМ по использованию ЦП, но также можно использовать различные параметры для сортировки по другим критериям, таким как использование памяти или сети.

Некоторые из ключевых параметров, которые можно использовать с xentop, включают:

-s: указать адрес хоста, на котором запущен гипервизор Xen.

-p: указать номер порта, используемый для подключения к гипервизору Xen (по умолчанию используется порт 26).

-u: указать имя пользователя для подключения к гипервизору Xen.

-d: указать интервал обновления списка ВМ в секундах.

-c: отображать информацию о использовании ЦП для каждого ядра (CPU core) гипервизора Xen.

-m: отображать информацию о использовании памяти для каждой ВМ.

Xentop является очень полезным инструментом для диагностики проблем с производительностью виртуальных машин в среде Xen и оптимизации использования ресурсов. Эта утилита доступна для большинства дистрибутивов Linux и может быть установлена с помощью стандартного менеджера пакетов. Однако, для ее работы может потребоваться настройка доступа к гипервизору Xen с использованием соответствующих прав доступа.

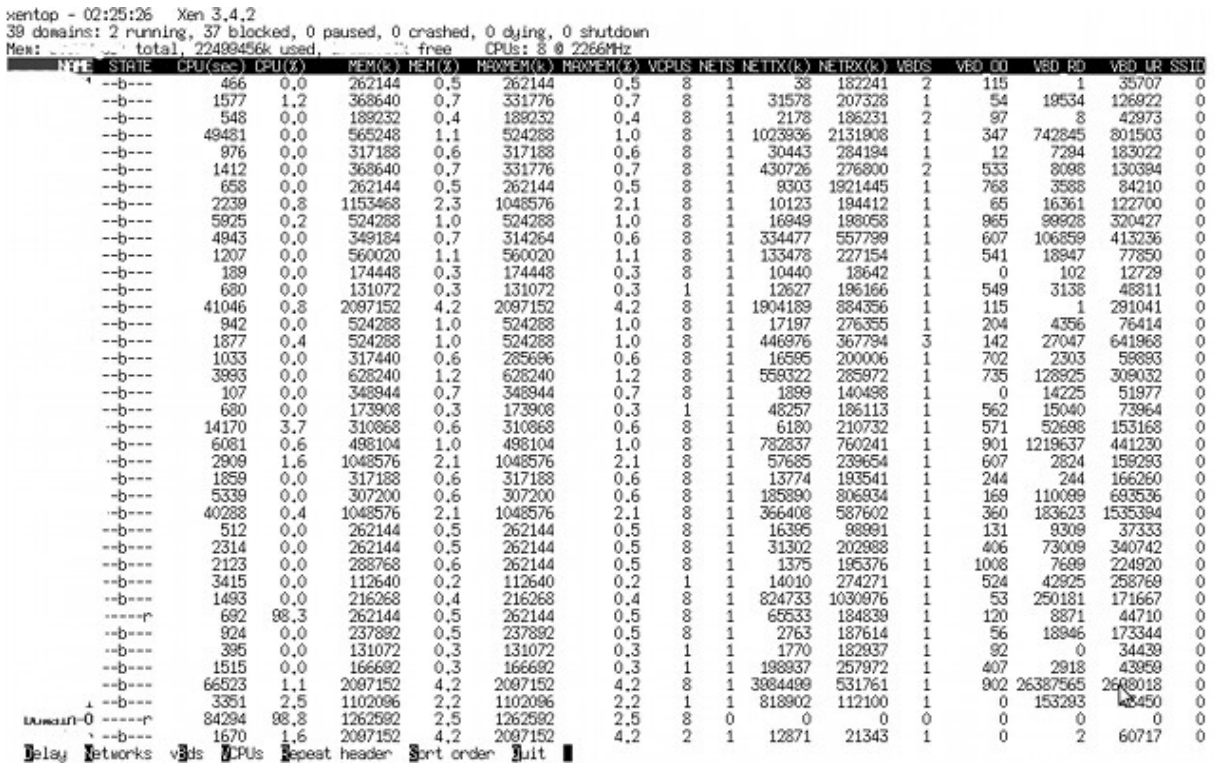


Рисунок 1.13 – Интерфейс утилиты xentop

1.3.14 nethogs

Nethogs - это утилита командной строки для мониторинга использования сетевого трафика отдельными процессами в операционной системе Linux. Эта

программа отображает список запущенных процессов и их использование сетевого трафика в реальном времени.

Nethogs работает путем анализа информации о сетевых пакетах, проходящих через сетевую карту, и сопоставления их с соответствующими процессами. Затем программа отображает список процессов, отсортированный по использованию сетевого трафика, и показывает статистику по переданным и полученным байтам, а также по текущей скорости передачи данных.

Некоторые из ключевых параметров, которые можно использовать с nethogs, включают:

-d: указать интервал обновления списка процессов в секундах.

-t: отображать статистику по переданным и полученным байтам в виде таблицы.

-p: фильтровать список процессов по указанному порту.

-u: отображать информацию о владельце процесса (UID).

-h: отобразить справку по использованию программы.

Nethogs является очень полезным инструментом для диагностики проблем с сетевым трафиком, вызванных отдельными процессами в системе. Эта утилита доступна для большинства дистрибутивов Linux и может быть установлена с помощью стандартного менеджера пакетов. Однако, для ее работы может потребоваться настройка доступа к сетевой карте с использованием соответствующих прав доступа.

PID	USER	PROGRAM	DEV	SENT	RECEIVED
3009	amarao	/usr/bin/python	eth2	3392.547	49.332 KB/sec
4458	www-data	/usr/sbin/apache2	eth2	18.372	1.497 KB/sec
31001	amarao	./el.x86_64.linux.bin	eth2	0.193	0.218 KB/sec
0	root	..6:80-85.118.226.108:38600		1.149	0.190 KB/sec
0	root	..6:80-109.110.40.176:49426		2.473	0.172 KB/sec
0	root	..2.76:80-109.254.49.8:3325		0.077	0.151 KB/sec
0	root	..6:80-88.204.125.101:59399		2.484	0.146 KB/sec
0	root	..6:80-85.118.226.108:56970		0.914	0.139 KB/sec
0	root	..57776-94.100.19.196:49462		0.178	0.129 KB/sec
0	root	..6:1c25:aed:2c1a:b318:6552		0.000	0.054 KB/sec
0	root	..57776-98.109.218.42:57671		0.033	0.048 KB/sec
0	root	..:57776-124.104.97.58:1443		0.033	0.048 KB/sec
0	root	..:57776-24.37.115.49:55605		0.033	0.048 KB/sec
0	root	..6:1c25:aed:2c1a:b318:6552		0.031	0.038 KB/sec
7171	root	pptp	eth2	0.024	0.026 KB/sec
4708	www-data	/usr/sbin/apache2	eth2	0.013	0.013 KB/sec
0	root	..49589-220.237.130.63:9904		0.011	0.012 KB/sec
0	root	..:57776-64.217.18.183:3600		0.000	0.012 KB/sec
4698	www-data	/usr/sbin/apache2	eth2	0.011	0.012 KB/sec
0	root	..:57776-64.217.18.183:3599		0.000	0.012 KB/sec
0	root	..:57776-184.56.20.44:54224		0.000	0.000 KB/sec
0	root	..6:80-209.121.54.212:59156		0.000	0.000 KB/sec
0	root	unknown TCP		0.000	0.000 KB/sec
TOTAL				3418.577	52.296 KB/sec

Рисунок 1.14 – Интерфейс утилиты nethogs

1.4 Сравнительный анализ

Top и htop являются двумя самыми популярными утилитами для мониторинга процессов в Unix-подобных операционных системах. Обе программы предоставляют пользователю информацию о процессах, запущенных в системе, и позволяют управлять ими. В этом разделе будет проведен сравнительный анализ этих двух программ, а также выделены важные моменты, которые необходимо учесть при реализации собственного проекта.

Top имеет текстовый интерфейс, который обновляется каждые несколько секунд. Он отображает список процессов в табличном виде, содержащем информацию о PID, пользователе, приоритете, использовании памяти и процессора, времени выполнения и других параметрах. Пользователь может отсортировать процессы по любому из этих параметров, а также фильтровать их по различным критериям.

Htop также имеет текстовый интерфейс, но он более визуальный и интуитивно понятный. Программа отображает список процессов в виде таблицы, где каждая строка содержит цветную индикацию использования ресурсов. Пользователь может прокручивать список вверх и вниз, а также использовать мышь для выделения процессов и выполнения действий с ними. Кроме того, htop предоставляет графическое представление использования процессора, памяти и swap-памяти.

Функциональность:

Top и htop предоставляют схожую функциональность, но есть некоторые отличия. Top предоставляет больше опций для настройки отображения информации о процессах, например, можно выбрать, какие столбцы отображать в таблице. Кроме того, top позволяет выполнять некоторые действия с процессами, такие как убийство процесса или изменение его приоритета.

Htop также предоставляет возможность управлять процессами, но он делает это более удобным способом. Например, пользователь может выделить несколько процессов и выполнить с ними одно действие, или просто нажать клавишу F9, чтобы убить выделенный процесс. Кроме того, htop предоставляет возможность отправлять сигналы процессам, например, `SIGTERM` или `SIGKILL`.

Top и htop имеют разную производительность. Top использует меньше ресурсов системы, так как он обновляет информацию о процессах каждые несколько секунд.

Htop обновляет информацию в реальном времени, что требует большего количества ресурсов системы.

Кроссплатформенность

Top доступен на большинстве Unix-подобных операционных систем, включая Linux, macOS и BSD.

Htop также доступен на большинстве этих систем, но его нет в стандартной поставке macOS.

На основе проведённого анализа следует выделить важные моменты, которые необходимо учесть при реализации собственного проекта:

Интерфейс должен быть интуитивно понятным и удобным для пользователя. Функциональность должна соответствовать потребностям пользователя и предоставлять необходимые возможности для управления процессами. Производительность должна быть оптимизирована для минимизации нагрузки на систему. Необходимо предусмотреть возможность настройки отображения информации о процессах в соответствии с потребностями пользователя. Предусмотреть возможность отправки сигналов процессам для управления ими. Предусмотреть возможность фильтрации процессов по различным критериям для удобства пользователя. Предусмотреть возможность сортировки процессов по различным параметрам для удобства пользователя. Предусмотреть возможность настройки интервала обновления информации о процессах для оптимизации производительности.

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Установка требований к функционалу разрабатываемой в рамках курсового проекта программе позволяет провести разделение всего алгоритма работы приложения на функциональные блоки. Функциональные блоки – это блоки программного компонента, которые ответственны за определенную задачу, а совокупность функциональных блоков позволяет реализовать полноценную работу программы. Наличие функциональных блоков сокращает количество времени на понимание внутреннего устройства программы, обеспечивая гибкость и масштабируемость приложения с целью последующей возможной доработки путем добавления дополнительных программных блоков.

Программу диспетчера процессов и потоков можно разделить на 6 функциональных блоков:

Блок ввода-вывода: этот блок отвечает за взаимодействие с пользователем и отображение информации о процессах и потоках. Он включает в себя функции для отображения списка процессов и потоков, обработки ввода пользователя и вывода сообщений об ошибках.

Блок чтения данных: этот блок отвечает за чтение данных о процессах и потоках из системных файлов. Он включает в себя функции для чтения информации из каталога /proc и заполнения структур `ProcessInfo` и `ThreadInfo`,

Блок управления процессами и потоками: этот блок отвечает за управление процессами и потоками, включая их завершение и сортировку по разным критериям. Он включает в себя функции для сортировки массивов структур `ProcessInfo` и `ThreadInfo` с помощью стандартной функции `qsort и` функцию для отправки сигналов процессам и потокам.

Блок главного цикла программы: этот блок отвечает за управление основным циклом программы, включая очистку экрана, обновление данных о процессах и потоках, отображение информации и обработку ввода пользователя.

Взаимодействие между этими блоками происходит следующим образом:

Блок ввода-вывода получает команды от пользователя и передает их в соответствующие блоки для обработки. Блок чтения данных читает информацию о процессах и потоках из системных файлов и заполняет структуры `ProcessInfo` и `ThreadInfo`. Блок сортировки сортирует массивы структур `ProcessInfo` и `ThreadInfo` в соответствии с критериями, заданными пользователем. Блок управления процессами и потоками отправляет сигналы процессам и потокам в соответствии с командами, полученными от блока ввода-вывода. Блок обработки сигналов обрабатывает сигналы, генерируемые операционной системой, и выполняет необходимые действия, такие как завершение программы при получении сигнала `SIGINT`. Блок главного цикла программы управляет основным циклом программы, вызывая функции из других блоков для обновления данных, отображения информации и обработки ввода пользователя.

2.1 Блок ввода-вывода

Блок ввода-вывода отвечает за взаимодействие с пользователем и отображение информации о процессах и потоках. Он предоставляет пользователю интерфейс для ввода команд и выводит результаты их выполнения. В частности, этот блок включает в себя функции для отображения списка процессов и потоков, обработки ввода пользователя и вывода сообщений об ошибках.

2.2 Блок чтения данных

Блок чтения данных отвечает за чтение данных о процессах и потоках из системных файлов. Он читает информацию из каталога /proc и заполняет структуры `ProcessInfo` и `ThreadInfo`. А также для преобразования этой информации в формат, подходящий для отображения и сортировки.

2.3 Блок управления процессами и потоками

Блок управления процессами и потоками отвечает за управление процессами и потоками, включая их завершение и сортировку по разным критериям, таким как идентификатор, использование физической и виртуальной памяти. Он предоставляет пользователю возможность управлять процессами и потоками, отправляя им соответствующие сигналы для их завершения.

2.4 Блок главного цикла программы

Блок главного цикла программы отвечает за управление основным циклом программы, включая очистку экрана, обновление данных о процессах и потоках, отображение информации и обработку ввода пользователя. Он обеспечивает взаимодействие между всеми другими блоками и управляет потоком выполнения программы. Блок включает в себя функции для очистки экрана, обновления данных о процессах и потоках, отображения информации и обработки ввода пользователя.

Каждый из этих блоков выполняет определенную задачу и взаимодействует с другими блоками для обеспечения полноценной работы программы. Наличие функциональных блоков позволяет сократить количество времени на понимание внутреннего устройства программы и обеспечить гибкость и масштабируемость приложения с целью последующей возможной доработки путем добавления дополнительных программных блоков.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе описывается структура разрабатываемой в рамках курсового проекта программы с точки зрения описания данных и обрабатывающих их подпрограмм – функций.

3.1 Описание основных структур данных программы

В программе используются следующие основные структуры данных:

Структура `ProcessInfo` используется для хранения основной информации о процессе.

```
typedef struct {
    int pid;
    char user[50];
    char state;
    double resident_memory;
    double virtual_memory;
    int cpu_cores;
    int threads;
    char start_time[20];
    char command[100];
} ProcessInfo;
```

`char user[50]` - имя пользователя, запустившего процесс. Это строковое поле, содержащее до 50 символов, указывает, под какой учетной записью был запущен процесс.

`char state` - состояние процесса. Этот символ указывает текущий статус процесса (например, 'R' для работающего, 'S' для спящего и т.д.).

`double resident_memory` - объем физической памяти, потребляемой процессом, в мегабайтах. Это значение показывает, сколько оперативной памяти использует процесс.

`double virtual_memory` - объем виртуальной памяти, используемой процессом, в мегабайтах. Виртуальная память включает в себя как физическую память, так и свопинг (использование жесткого диска).

`int cpu_cores` - количество ядер процессора, используемых процессом. Это поле показывает, насколько процесс нагружает систему.

`int threads` - количество потоков, запущенных процессом. Процессы могут состоять из нескольких потоков, которые выполняются параллельно.

`char start_time[20]` - время запуска процесса. Это строковое поле фиксированной длины указывает, когда процесс был запущен.

`char command[100]` - имя команды, запустившей процесс. Это строка длиной до 100 символов содержит полное имя или путь к исполняемому файлу.

Структура ThreadInfo используется для хранения информации о потоках процесса.

```
typedef struct {
    int tid;
    char state;
    char name[16];
} ThreadInfo;
```

int tid - идентификатор потока (TID). Подобно PID, но уникален для потоков внутри процесса.

char state - состояние потока. Этот символ указывает текущий статус потока (например, 'R' для работающего, 'S' для спящего и т.д.).

char name[16] - имя потока. Это строковое поле длиной до 16 символов может содержать имя или описание потока.

Структура ProcessData объединяет информацию о процессе и его потоках.

```
typedef struct {
    ProcessInfo process_info;
    ThreadInfo threads[MAX_THREADS_PER_PROCESS];
    int thread_count;
} ProcessData;
```

ProcessInfo process_info - структура ProcessInfo, содержащая информацию о процессе.

ThreadInfo threads[MAX_THREADS_PER_PROCESS] - массив структур ThreadInfo, содержащий информацию о потоках процесса. Максимальное количество потоков на процесс определяется константой MAX_THREADS_PER_PROCESS, которая в данном случае равна 100.

int thread_count - количество потоков в данном процессе. Это поле указывает, сколько элементов массива threads задействовано.

Эти структуры данных используются для хранения информации о процессах и потоках в программе. Они являются основой для реализации функционала программы, такого как отображение списка процессов и потоков, сортировка списка, управление процессами и потоками, и т.д.

3.2 Описание основных функций программы

3.2.1 Файл control.c

`void kill_process_or_thread()` - а функция отвечает за завершение процесса или потока по его идентификатору (PID или TID). Включает отображение вводимых символов и курсора. Устанавливает таймаут на ожидание ввода в 15 секунд. Запрашивает у пользователя ввод PID или TID для завершения. Если введено значение больше 0, пытается завершить процесс/поток с этим ID. Сообщает об успехе или неудаче операции. После завершения отключает отображение символов и курсор, очищает экран.

`int compare_by_pid(const void *a, const void *b)`- а функция используются для сортировки процессов по их PID.

`int compare_by_resident_memory(const void *a, const void *b)`- функция используются для сортировки процессов по объему занятой оперативной памяти.

`int compare_by_virtual_memory (const void *a, const void *b)`- функция используются для сортировки процессов по объему занятой виртуальной памяти.

`void handle_user_input(int ch)` – функция обрабатывает ввод пользователя и выполняет соответствующие действия.

`'p'` или `'P'`: Изменяет критерий сортировки на PID или меняет порядок сортировки.

`'r'` или `'R'`: Изменяет критерий сортировки на объем занятой оперативной памяти или меняет порядок сортировки.

`'v'` или `'V'`: Изменяет критерий сортировки на объем занятой виртуальной памяти или меняет порядок сортировки.

`'k'` или `'K'`: Вызывает функцию `kill_process_or_thread()` для завершения процесса/потока.

`'q'` или `'Q'`: Завершает программу.

Другие клавиши: Игнорируются.

3.2.2 Файл display.c

`Void display_thread_info(const ThreadInfo *thread_info)` – функция отвечает за отображение информации о потоке. Она принимает указатель на структуру `ThreadInfo`, которая содержит различные данные о потоке, такие как идентификатор потока (TID), его состояние и имя. Внутри функции эти данные форматируются в строку и выводятся в виде строки таблицы. Каждая строка таблицы содержит информацию о конкретном потоке, такую как его идентификатор, состояние и имя.

`void display_header(ColorScheme color_scheme)` – функция отвечает за отображение заголовка таблицы с информацией о процессах. Перед отображением заголовка она получает текущие размеры экрана и цветовую схему. Затем форматирует и выводит текущую дату и время в центре верхней строки. После этого отображает заголовки столбцов таблицы. Заголовки столбцов включают PID процесса, пользователя, состояние процесса, объем используемой оперативной памяти (resident memory), объем виртуальной памяти (virtual memory), количество ядер процессора, время запуска и команду запуска.

`void display_process_info(const ProcessInfo *proc_info)` – функция отображает информацию о процессе. Она принимает указатель на структуру `ProcessInfo`, содержащую данные о процессе, такие как его PID, имя пользователя, состояние, объем используемой оперативной и виртуальной памяти, количество ядер процессора, время запуска и команда запуска. Функция форматирует эти данные и выводит их в виде строки таблицы, соответствующей формату, установленному в заголовке таблицы.

`void update_display(int start_line, int total_lines, ProcessData *process_data, int process_count, DisplayMode mode, ColorScheme color_scheme)` – функция обновляет отображение информации о процессах и потоках на экране. Она сортирует данные перед отображением в соответствии с текущим критерием сортировки. Затем очищает экран и выводит заголовок таблицы. После этого она поочередно выводит информацию о каждом процессе и его потоках в таблицу на экране. Функция учитывает текущий режим отображения (процессы или процессы и потоки) и цветовую схему при выводе информации.

3.2.3 Файл main.c

`int main()` – функция начинается с инициализации библиотеки ncurses и установки основных параметров, таких как поддержка цвета и отключение эха ввода. Затем она входит в бесконечный цикл, который обновляет отображение информации о процессах и потоках. Извлекает информацию о процессе, такую как его PID, состояние и использование памяти. Обновляет отображение на экране в соответствии с текущими данными. Обрабатывает ввод пользователя, такой как прокрутка и изменение режима отображения, вывод справки, чтобы обеспечить интерактивность пользовательского интерфейса.

3.2.4 Файл read.c

`void get_thread_info(ThreadInfo *thread_info, int pid, int tid)` – функция отвечает за получение информации о потоке. Она принимает указатель на структуру `ThreadInfo`, а также идентификаторы процесса (pid) и потока (tid). Сначала функция формирует путь к файлам с

информацией о потоке в /proc, затем открывает файлы и извлекает необходимые данные, такие как имя потока и его состояние. Полученная информация сохраняется в переданной структуре `ThreadInfo`.

`int count_cpu_cores(const char *cpu_list)` – функция используется для подсчета количества ядер процессора, доступных для процесса. Она принимает строку, представляющую список доступных ядер процессора, и возвращает количество ядер.

`time_t get_system_uptime()` – функция используется для получения времени работы системы. Она открывает файл /proc/uptime, извлекает из него значение времени работы системы и возвращает его в виде времени типа time_t.

`void get_process_info(ProcessInfo *proc_info, int pid)` – функция отвечает за получение информации о процессе. Она принимает указатель на структуру `ProcessInfo` и идентификатор процесса (pid). Функция сначала читает информацию из файлов /proc/[pid]/status и /proc/[pid]/stat, извлекая такие данные, как имя пользователя, состояние процесса, объем используемой памяти и команду. Затем она вычисляет дополнительные параметры, такие как количество ядер процессора, количество потоков, объем физической памяти и время запуска процесса. Полученная информация сохраняется в переданной структуре `ProcessInfo`.

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

В данном разделе представлены схемы алгоритмов и алгоритмы по шагам основных функций разработанной в рамках курсового проекта.

4.1 Разработка структурной схемы

Структурная схема программы приведена в приложении А.

4.2 Схемы алгоритмов

4.2.1 Схема алгоритма `get_thread_info`

Функция получает информацию о конкретном потоке в процессе. Она принимает на вход идентификатор процесса (PID) и идентификатор потока (TID). Затем она открывает соответствующие файлы в каталоге /proc/[pid]/task/[tid] для чтения информации о потоке. Схема алгоритма `get_thread_info` приведена в приложении Б.

4.2.2 Схема алгоритма `count_cpu_cores`

Функция подсчитывает количество ядер процессора, на которых может выполняться процесс, исходя из списка ядер, на которых разрешено выполнение (список `cpu_list`). Она принимает этот список в качестве входного параметра и возвращает количество ядер. Функция сканирует список и подсчитывает количество запятых, что позволяет определить число ядер. Схема алгоритма `count_cpu_cores` приведена в приложении В.

4.3 Разработка алгоритмов

4.3.1 Алгоритм функции `kill_process_or_thread`

Функция `kill_process_or_thread` обеспечивает пользователю возможность завершения процесса или потока. Шаги выполнения этой функции:

1. Включение отображения вводимых символов и курсора: Функция вызывает `echo()` для включения отображения вводимых символов и `curs_set(1)` для включения курсора.
2. Установка времени ожидания ввода на 15 секунд: Используется `timeout(15000)` для установки времени ожидания ввода на 15 секунд.
3. Ввод идентификатора (TID или PID): Пользователю предлагается ввести идентификатор процесса или потока.
4. Получение идентификатора и преобразование в целое число: Введенная строка преобразуется в целое число с помощью `atoi()`.

5. Проверка на выход: Если введенное значение равно 0, функция завершает свою работу.

6. Попытка завершения процесса или потока: Вызывается функция `kill()` для отправки сигнала `SIGKILL` процессу или потоку с введенным идентификатором.

7. Вывод результата попытки завершения: На экран выводится сообщение о результате попытки завершения.

8. Повторение процесса ввода и попытки завершения: Цикл повторяется, пока время ожидания не истекло или пока пользователь не введет 0.

9. Выключение отображения вводимых символов и курсора: После завершения работы цикла функция отключает отображение вводимых символов и курсор.

10. Очистка экрана и завершение функции: Экран очищается с помощью clear(), и функция завершает свою работу.

4.3.2 Алгоритм функции `get_process_info`

Функция `read_processes()` отвечает за чтение информации о процессах и сохранение ее в структуре `Process`. Она читает информацию из файла /proc/[pid]/status, где [pid] - идентификатор процесса. Шаги выполнения этой функции:

1. Получение информации о процессе:
Функция получает в качестве аргументов указатель на структуру `ProcessInfo` и идентификатор процесса (pid).

2. Инициализация переменных и открытие файлов:
Инициализируются переменные `path` и `buffer`. Открывается файл /proc/[pid]/status для чтения информации о процессе.

3. Чтение информации из /proc/[pid]/status:
В цикле while считывается каждая строка из файла. Если строка начинается с `Uid`:, из нее извлекается идентификатор пользователя (`uid`), который используется для получения имени пользователя с помощью функции `getpwuid()`. Если строка начинается с `VmSize`:, из нее извлекается размер виртуальной памяти процесса (`vm_size`). Если строка начинается с `State`:, из нее извлекается состояние процесса (`state`). Если строка начинается с `Name`:, из нее извлекается имя исполняемого файла процесса (`command`). Если строка начинается с `Cpus_allowed_list`:, из нее извлекается строка, содержащая список доступных ядер процессора, из которой определяется количество ядер (`cpu_cores`). Если строка начинается с `Threads`:, из нее извлекается количество потоков (`threads`).

4. Чтение информации из /proc/[pid]/stat:

Открывается файл /proc/[pid]/stat для получения информации о времени работы процесса и потреблении физической памяти. Считываются необходимые значения: `utime`, `stime`, `starttime`, `rss`.

5. Расчет времени запуска процесса:
Вычисляется текущее время в секундах (`now`). Получается время работы системы в секундах (`uptime`) с помощью функции `get_system_uptime()`. Вычисляется время запуска процесса (`start_time`). Для форматирования времени используется функция `localtime_r()`.

6. Вычисление объема занимаемой физической памяти:
Открывается файл /proc/[pid]/statm. Считывается количество страниц резидентной памяти (`rss`), занимаемых процессом. Вычисляется объем физической памяти в мегабайтах (`resident_memory`).

7. Заполнение структуры `ProcessInfo`:
Полученные данные записываются в поля структуры `ProcessInfo`.

8. Закрытие файлов:
Все открытые файлы закрываются.

9. Завершение работы функции:

4.4 Код программы

Код программы представлен в приложении Г.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

5.1 Требования к программному и аппаратному обеспечению

Процессор: любой совместимый с архитектурой x86 или x86-64
Оперативная память: не менее 128 МБ
Жесткий диск: не менее 10 МБ свободного места

5.2 Руководство по использованию

При запуске программы пользователю будет выведен заголовок со столбцами PID, USER, STATE, RES_MEM, VIRT_MEM, CORES, START, COMMAND и заголовок с текущими датой и временем, под столбцами заголовка выведена информация для каждого из процессов.

Kustrica 2024-05-26 23:16:08

PID	USER	STATE	RES_MEM	VIRT_MEM	CORES	START	COMMAND
1	root	S	14.0 MB	0.0 MB	1	14:45:52	systemd
2	root	S	0.0 MB	0.0 MB	1	14:45:52	kthreadd
3	root	S	0.0 MB	0.0 MB	1	14:45:52	pool_workqueue_release
4	root	I	0.0 MB	0.0 MB	1	14:45:52	kworker
5	root	I	0.0 MB	0.0 MB	1	14:45:52	kworker
6	root	I	0.0 MB	0.0 MB	1	14:45:52	kworker
7	root	I	0.0 MB	0.0 MB	1	14:45:52	kworker
9	root	I	0.0 MB	0.0 MB	1	14:45:52	kworker
12	root	I	0.0 MB	0.0 MB	1	14:45:52	kworker
13	root	I	0.0 MB	0.0 MB	1	14:45:52	rcu_tasks_kthread
14	root	I	0.0 MB	0.0 MB	1	14:45:52	rcu_tasks_rude_kthread
15	root	I	0.0 MB	0.0 MB	1	14:45:52	rcu_tasks_trace_kthread
16	root	S	0.0 MB	0.0 MB	1	14:45:52	ksoftirqd
17	root	I	0.0 MB	0.0 MB	1	14:45:52	rcu_preempt
18	root	S	0.0 MB	0.0 MB	1	14:45:52	migration
19	root	S	0.0 MB	0.0 MB	1	14:45:52	idle_inject
20	root	S	0.0 MB	0.0 MB	1	14:45:52	cpuhp
21	root	S	0.0 MB	0.0 MB	1	14:45:52	cpuhp
22	root	S	0.0 MB	0.0 MB	1	14:45:52	idle_inject
23	root	S	0.0 MB	0.0 MB	1	14:45:52	migration
24	root	S	0.0 MB	0.0 MB	1	14:45:52	ksoftirqd
26	root	I	0.0 MB	0.0 MB	1	14:45:52	kworker

Рисунок 5.2.1 – Список процессов

Функции, которые пользователь может вызвать нажатием клавиш:
По нажатию на клавиатуре клавиши t или T, раскроется древовидный список потоков. Под каждым и процессов будут показаны его потоки и их идентификаторы.

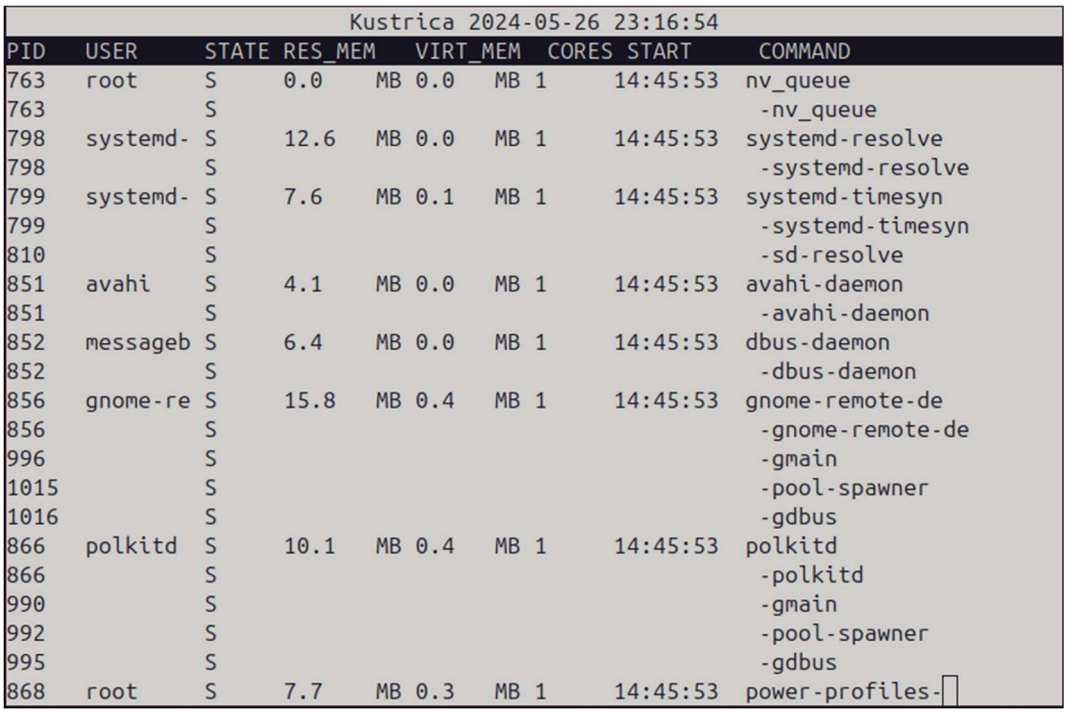


Рисунок 5.2.2 – Список процессов и потоков

По нажатию на клавиатуре клавиши p или P, произойдет сортировка процессов по PID по возрастанию.

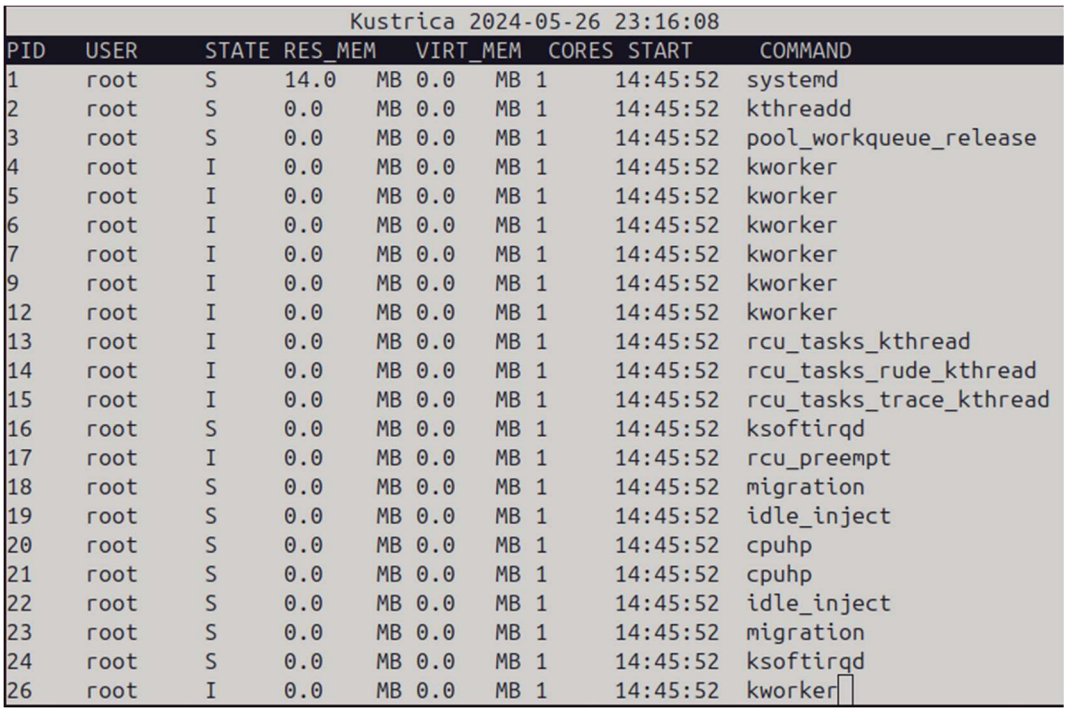


Рисунок 5.2.3 – Список процессов отсортированный по возрастанию PID

По нажатию на клавиатуре клавиши p или P повторно, произойдет сортировка процессов по PID по убыванию.

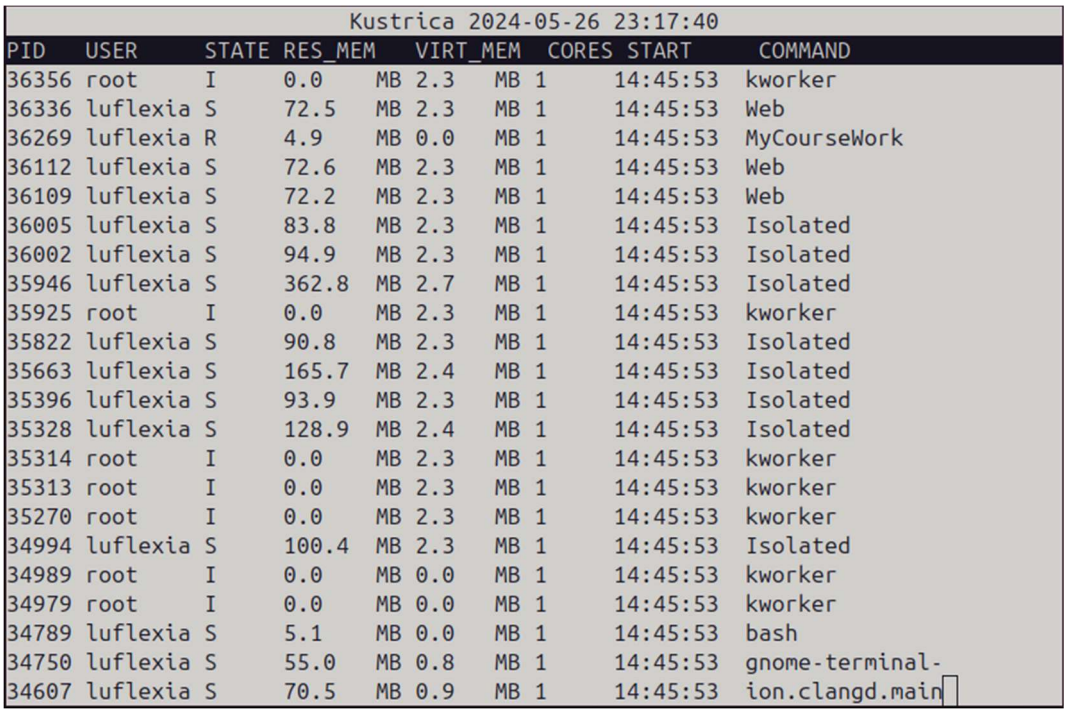


Рисунок 5.2.4 – Список процессов отсортированный по убыванию PID

По нажатию на клавиатуре клавиши r или R, произойдет сортировка процессов по RES_MEM по убыванию.

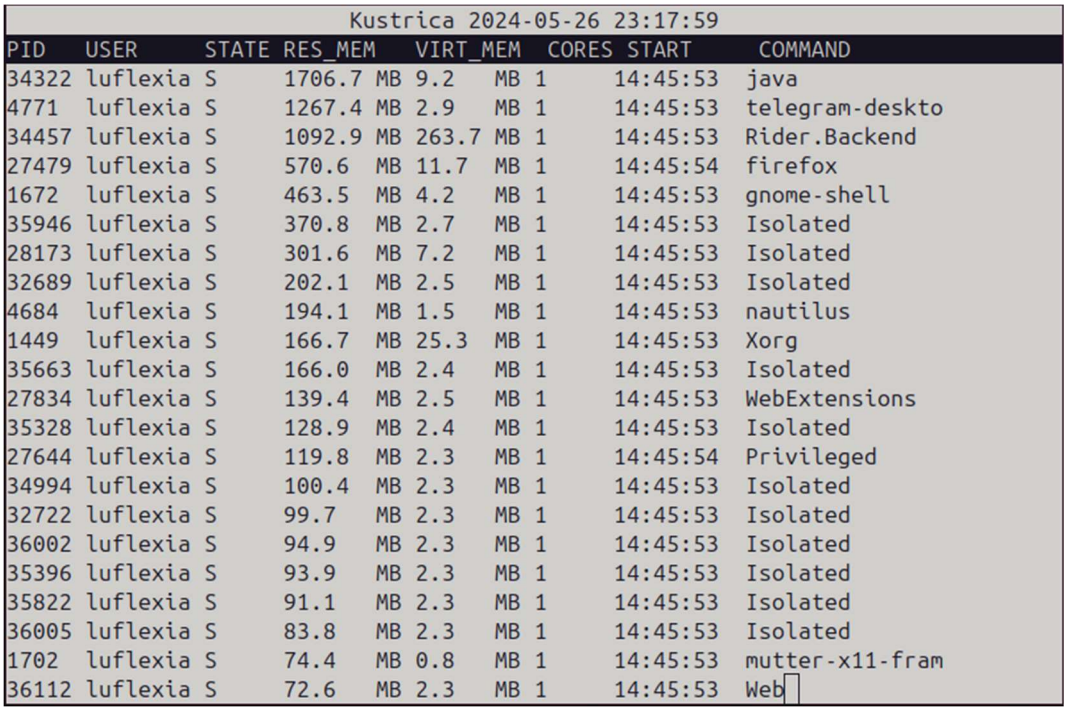
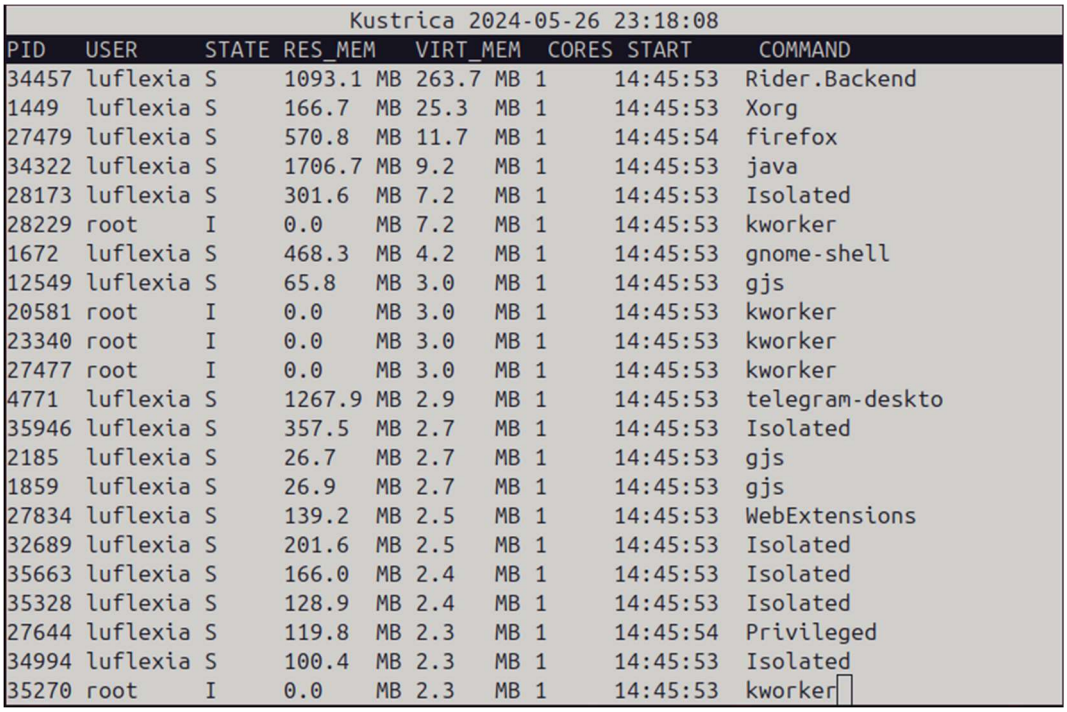


Рисунок 5.2.5 – Список процессов отсортированный по убыванию RES_MEM

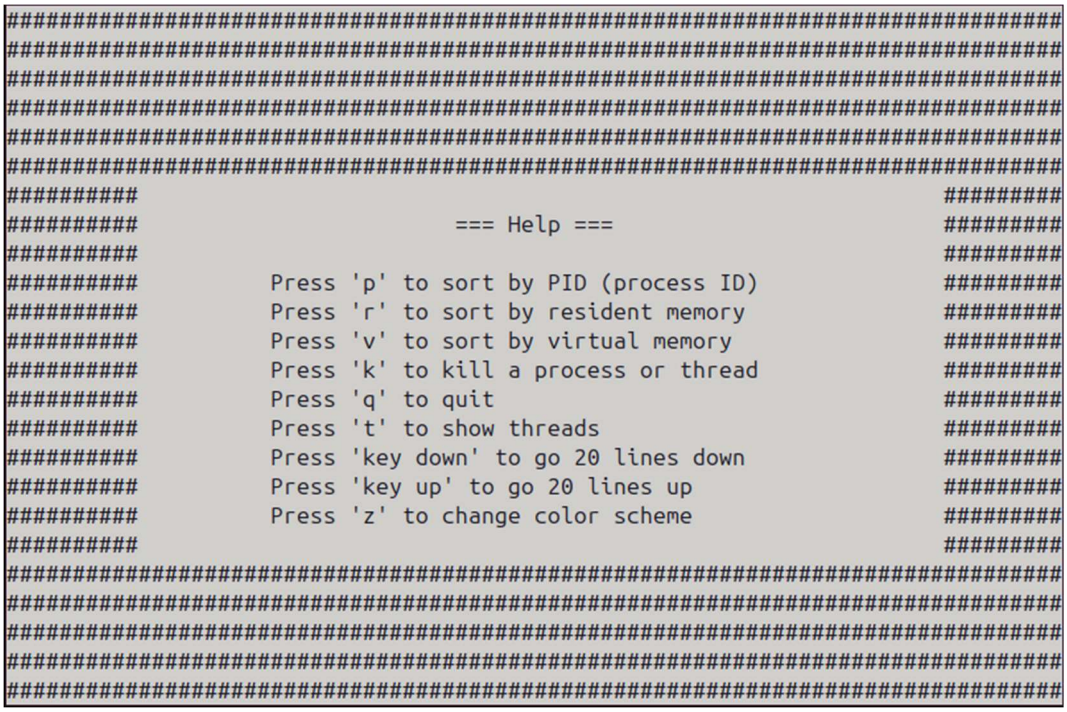
По нажатию на клавиатуре клавиши v или V, произойдет сортировка процессов по VIRT_MEM по убыванию.



```
                                Kustrica 2024-05-26 23:18:08
PID   USER     STATE RES_MEM   VIRT_MEM  CORES START     COMMAND
34457 luflexia S     1093.1 MB 263.7 MB 1     14:45:53  Rider.Backend
1449  luflexia S     166.7  MB 25.3  MB 1     14:45:53  Xorg
27479 luflexia S     570.8  MB 11.7  MB 1     14:45:54  firefox
34322 luflexia S     1706.7 MB 9.2   MB 1     14:45:53  java
28173 luflexia S     301.6  MB 7.2   MB 1     14:45:53  Isolated
28229 root     I     0.0    MB 7.2   MB 1     14:45:53  kworker
1672  luflexia S     468.3  MB 4.2   MB 1     14:45:53  gnome-shell
12549 luflexia S     65.8   MB 3.0   MB 1     14:45:53  gjs
20581 root     I     0.0    MB 3.0   MB 1     14:45:53  kworker
23340 root     I     0.0    MB 3.0   MB 1     14:45:53  kworker
27477 root     I     0.0    MB 3.0   MB 1     14:45:53  kworker
4771  luflexia S     1267.9 MB 2.9   MB 1     14:45:53  telegram-deskto
35946 luflexia S     357.5  MB 2.7   MB 1     14:45:53  Isolated
2185  luflexia S     26.7   MB 2.7   MB 1     14:45:53  gjs
1859  luflexia S     26.9   MB 2.7   MB 1     14:45:53  gjs
27834 luflexia S     139.2  MB 2.5   MB 1     14:45:53  WebExtensions
32689 luflexia S     201.6  MB 2.5   MB 1     14:45:53  Isolated
35663 luflexia S     166.0  MB 2.4   MB 1     14:45:53  Isolated
35328 luflexia S     128.9  MB 2.4   MB 1     14:45:53  Isolated
27644 luflexia S     119.8  MB 2.3   MB 1     14:45:54  Privileged
34994 luflexia S     100.4  MB 2.3   MB 1     14:45:53  Isolated
35270 root     I     0.0    MB 2.3   MB 1     14:45:53  kworker
```

Рисунок 5.2.6 – Список процессов отсортированный по убыванию VIRT_MEM

По нажатию на клавиатуре клавиши h или H, откроется справка, которая покажет функциональные клавиши, выйти из нее можно нажав любую клавишу.



```
################################################################################
################################################################################
################################################################################
################################################################################
################################################################################
################################################################################
##########                                                            #########
##########                    === Help ===                            #########
##########                                                            #########
##########         Press 'p' to sort by PID (process ID)              #########
##########         Press 'r' to sort by resident memory               #########
##########         Press 'v' to sort by virtual memory                #########
##########         Press 'k' to kill a process or thread              #########
##########         Press 'q' to quit                                  #########
##########         Press 't' to show threads                          #########
##########         Press 'key down' to go 20 lines down               #########
##########         Press 'key up' to go 20 lines up                   #########
##########         Press 'z' to change color scheme                   #########
##########                                                            #########
################################################################################
################################################################################
################################################################################
################################################################################
################################################################################
```

Рисунок 5.2.7 – Справка в светлом режиме отображения

По нажатию на клавиатуре клавиши z или Z, цвета в консоли инвертируются, фон станет черным, а цвет текста белым.

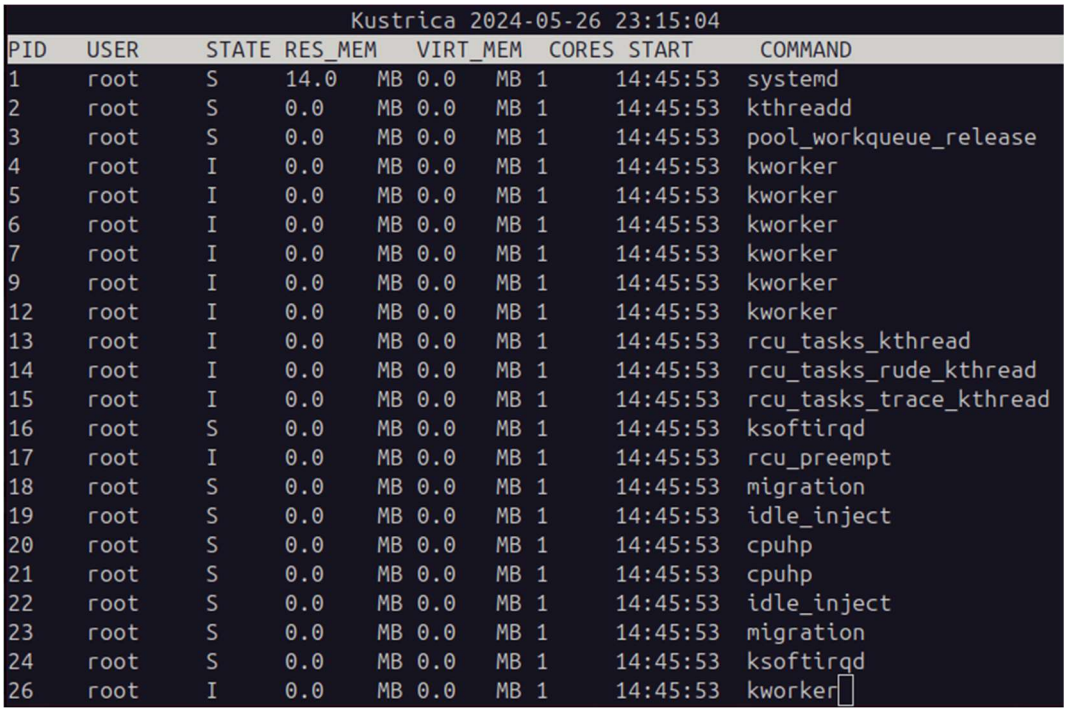


Рисунок 5.2.8 – Список процессов в темном режиме отображения

По нажатию на клавиатуре клавиши h или H, в темном режиме откроется справка, которая покажет функциональные клавиши.

Рисунок 5.2.9 – Справка в темном режиме отображения

По нажатию на клавиатуре клавиши k или K, на строке с заголовком появится поле для ввода идентификатор потока или процесса.

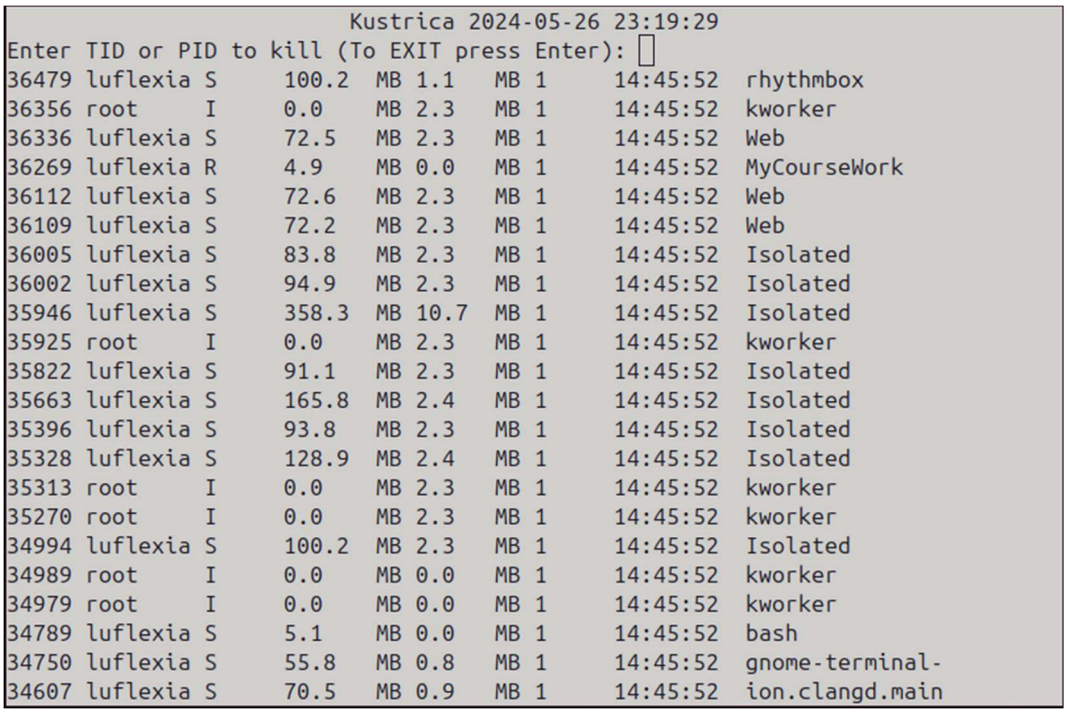


Рисунок 5.2.10 – Поле для ввода идентификатора потока или процесса

При попытке завершить процесс или поток, в случае успеха будет выведено соответствующее сообщение.

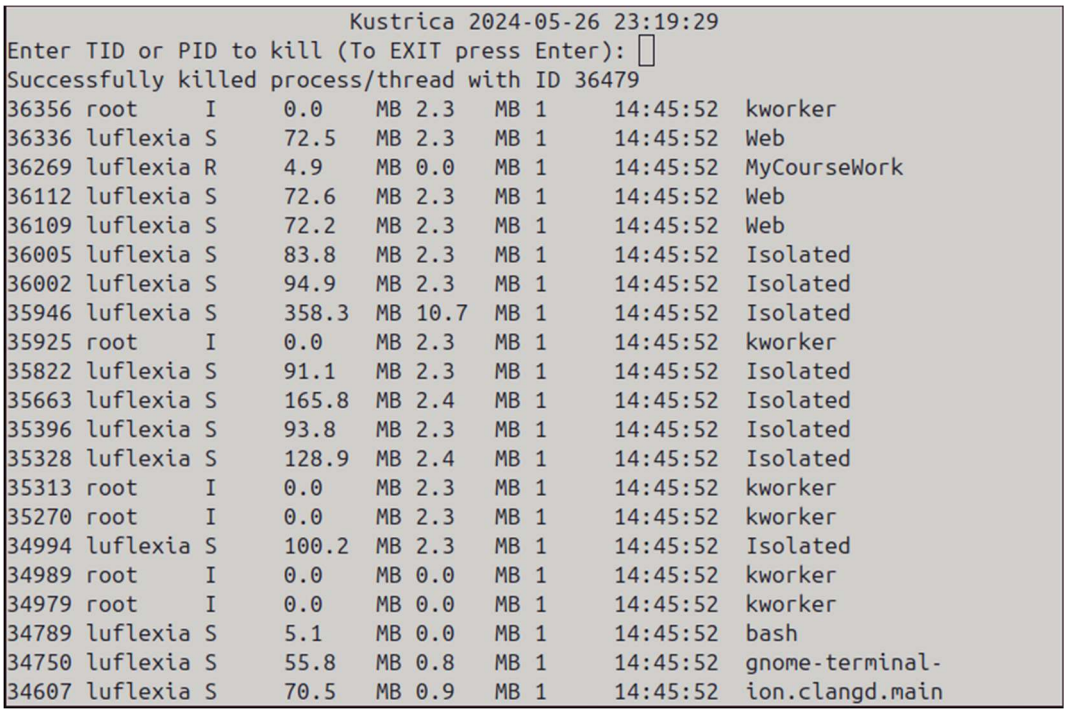


Рисунок 5.2.11 – Успешное завершение потока или процесса

В случае если процесс или поток завершить не удалось выведется соответствующее сообщение.

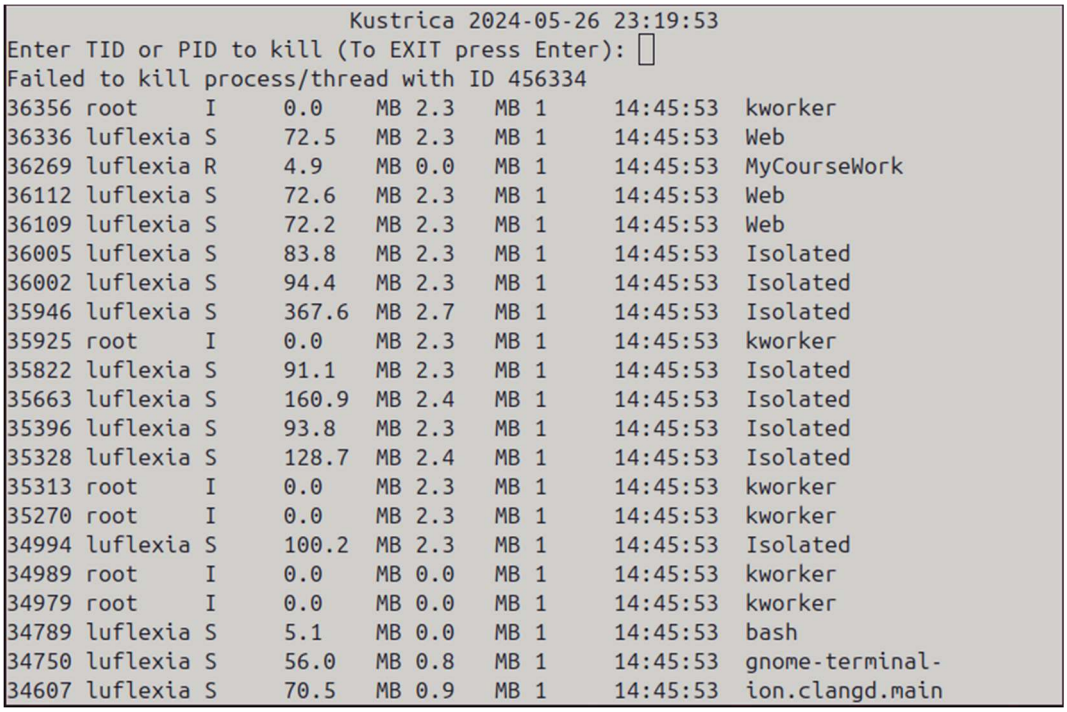


Рисунок 5.2.12 – Неудачное завершение потока или процесса

ЗАКЛЮЧЕНИЕ

В рамках курсового проекта была разработана программа диспетчера процессов и потоков, представляющая собой аналог утилиты top, которая предоставляет пользователю важный инструмент для мониторинга, анализа и мониторинга потоков и процессов, запущенных в системе.

Разработка диспетчера процессов и потоков включает в себя рассмотрение основных этапов, таких как получение списка процессов и потоков, обновление информации о них и отображение на экране. В процессе разработки необходимо было изучить системные вызовы и функции для работы с процессами и потоками операционной системы.

Программа предоставляет пользователю информацию о процессах, такую как идентификатор процесса (PID\TID), пользователь, потребление физической и виртуальной памяти, количество ядер процессора, которые использует процесс, какой командой был запущен процесс или поток. Кроме того, пользователь может сортировать и управлять процессами и потоками, что позволяет эффективно мониторить и анализировать работу системы.

Разработанный диспетчер процессов и потоков предоставляет пользователю возможность наблюдать текущие процессы, анализировать их характеристики и принимать решения на основе полученных данных. Это позволяет оптимизировать работу системы, выявлять и устранять проблемы с производительностью, а также обеспечивать безопасность системы.

В процессе разработки был использован язык программирования Си, а также библиотеки для работы с процессами и потоками, а так же библиотека ncurses, которая предоставляет набор функций для создания текстовых пользовательских интерфейсов (TUI) в терминальном окне. Были реализованы такие функции, как получение списка процессов и потоков, обновление информации о них, сортировка процессов, обработка пользовательского ввода, а также управление процессами и потоками.

Программа может быть дополнена и расширена для поддержки дополнительных функциональных возможностей, таких как мониторинг сетевой активности, анализ дискового пространства и мониторинг температуры компонентов системы.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

Брайан Керниган, Деннис Ритчи. Язык программирования Си. Издательство: «Вильямс», 2019 г.
Ричард Стивенс, Стивен Раго. UNIX. Профессиональное программирование. Издательство: «Вильямс», 2017 г.
Ричард Стивенс, Стивен Раго. Разработка приложений для UNIX. Издательство: «Питер», 2011 г.
The C Programming Language. Издательство: Prentice Hall, 1988 г.
top, htop, atop определение загрузки ОС (Load average, LA) [Электронный ресурс] – Режим доступа: https://wiki.dieg.info/top
Analysis with top in Linux [Электронный ресурс] – Режим доступа: https://prowse.tech/top/

ПРИЛОЖЕНИЕ А
(Обязательное)

Схема структурная

ПРИЛОЖЕНИЕ Б
(Обязательное)

Схема алгоритма `get_thread_info`

ПРИЛОЖЕНИЕ В
(Обязательное)

Схема алгоритма `count_cpu_cores`

ПРИЛОЖЕНИЕ Г
(Обязательное)

Код программы

Файл control.c

```
#include <curses.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include "display.h"
#include "control.h"

SortCriteria current_sort = SORT_BY_PID; // начальное значение
SortOrder sort_order = SORT_ORDER_ASCENDING; // начальный порядок сортировки


void kill_process_or_thread() {
    char input[10];
    int id = -1;

    echo(); // Включаем отображение вводимых символов
    curs_set(1); // Включаем курсор
    timeout(15000); // Устанавливаем время ожидания ввода 15 секунд

    while (1) {
        // Очищаем строку ввода и выводим приглашение к вводу
        move(1, 0);
        clrtoeol();
        printw("Enter TID or PID to kill (To EXIT press Enter): ");
        refresh();

        // Ввод ID
        if (getnstr(input, sizeof(input) - 1) == ERR) {
            // Если время истекло и пользователь не ввел ID, выходим из функции
            break;
        }
        id = atoi(input);

        // Проверка на выход
        if (id == 0) {
            break;
        }

        // Попытка убить процесс/поток
        move(2, 0); // Перемещаем курсор под строку ввода
        clrtoeol(); // Очищаем строку

        if (id > 0 && kill(id, SIGKILL) == 0) {
            printw("Successfully killed process/thread with ID %d\n", id);
        } else {
            printw("Failed to kill process/thread with ID %d\n", id);
        }
        refresh();
    }
    noecho(); // Отключаем отображение вводимых символов
    curs_set(0); // Выключаем курсор
    clear(); // Очищаем экран после завершения режима
    refresh();
}

int compare_by_pid(const void *a, const void *b) {
    const ProcessData *p1 = (const ProcessData *)a;
    const ProcessData *p2 = (const ProcessData *)b;
    return (sort_order == SORT_ORDER_ASCENDING) ?
           (p1->process_info.pid - p2->process_info.pid) :
           (p2->process_info.pid - p1->process_info.pid);
}

int compare_by_resident_memory(const void *a, const void *b) {
    const ProcessData *p1 = (const ProcessData *)a;
    const ProcessData *p2 = (const ProcessData *)b;
```

```
    if (p1->process_info.resident_memory < p2->process_info.resident_memory)
        return (sort_order == SORT_ORDER_ASCENDING) ? -1 : 1;
    if (p1->process_info.resident_memory > p2->process_info.resident_memory)
        return (sort_order == SORT_ORDER_ASCENDING) ? 1 : -1;
    return 0;
}

int compare_by_virtual_memory(const void *a, const void *b) {
    const ProcessData *p1 = (const ProcessData *)a;
    const ProcessData *p2 = (const ProcessData *)b;
    if (p1->process_info.virtual_memory < p2->process_info.virtual_memory)
        return (sort_order == SORT_ORDER_ASCENDING) ? -1 : 1;
    if (p1->process_info.virtual_memory > p2->process_info.virtual_memory)
        return (sort_order == SORT_ORDER_ASCENDING) ? 1 : -1;
    return 0;
}

void handle_user_input(int ch) {
    switch (ch) {
        case 'p':
            case 'P':
            if (current_sort == SORT_BY_PID) {
                sort_order = (sort_order == SORT_ORDER_ASCENDING) ? SORT_ORDER_DESCENDING :
SORT_ORDER_ASCENDING;
            } else {
                current_sort = SORT_BY_PID;
                sort_order = SORT_ORDER_ASCENDING;
            }
        break;
        case 'r':
            case 'R':
            if (current_sort == SORT_BY_RESIDENT_MEMORY) {
                sort_order = (sort_order == SORT_ORDER_ASCENDING) ? SORT_ORDER_DESCENDING :
SORT_ORDER_ASCENDING;
            } else {
                current_sort = SORT_BY_RESIDENT_MEMORY;
                sort_order = SORT_ORDER_ASCENDING;
            }
        break;
        case 'v':
            case 'V':
            if (current_sort == SORT_BY_VIRTUAL_MEMORY) {
                sort_order = (sort_order == SORT_ORDER_ASCENDING) ? SORT_ORDER_DESCENDING :
SORT_ORDER_ASCENDING;
            } else {
                current_sort = SORT_BY_VIRTUAL_MEMORY;
                sort_order = SORT_ORDER_ASCENDING;
            }
        break;
        case 'k':
            case 'K':
            kill_process_or_thread();
        break;
        case 'q':
            case 'Q':
            endwin();
        exit(0);
        default:
            break;
    }
}
```

Файл control.h

```
#ifndef CONTROL_H
#define CONTROL_H

typedef enum {
    SORT_BY_PID,
    SORT_BY_RESIDENT_MEMORY,
    SORT_BY_VIRTUAL_MEMORY
} SortCriteria;

typedef enum {
    SORT_ORDER_ASCENDING,
    SORT_ORDER_DESCENDING
```

```
} SortOrder;

extern SortCriteria current_sort;

void handle_user_input(int ch);
void kill_process_or_thread();
int compare_by_pid(const void *a, const void *b);
int compare_by_resident_memory(const void *a, const void *b);
int compare_by_virtual_memory(const void *a, const void *b);
#endif //CONTROL_H
```

Файл display.c

```
#include "display.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ncurses.h>
#include <unistd.h>
#include "processes.h"
#include "threads.h"
#include "control.h"

void display_thread_info(const ThreadInfo *thread_info) {
    printw("%-5d %-8s %-5c %-9s %-9s %-4s %-10s -%s\n",
           thread_info->tid, "", thread_info->state, "", "", "", "", thread_info->name);
}

void display_header(ColorScheme color_scheme) {
    int max_y, max_x; // Объявляем переменные для хранения размеров экрана
    getmaxyx(stdscr, max_y, max_x); // Получаем размеры окна
    char datetime[50];
    time_t rawtime;
    struct tm *timeinfo;

    time(&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(datetime, 50, "%Y-%m-%d %H:%M:%S", timeinfo); // Форматируем дату и время

    int datetime_length = strlen(datetime);
    const char *prefix = "Kustrica ";
    int prefix_length = strlen(prefix);
    int total_length = prefix_length + datetime_length;
    int padding = (max_x - total_length) / 2;

    // Выводим текст "Kustrica " и дату/время по центру первой строки без цветовой пары
    mvprintw(0, padding, "%s%s", prefix, datetime);

    // Устанавливаем цветовую пару для второй строки в зависимости от схемы
    if (color_scheme == COLOR_SCHEME_INVERTED) {
        attron(COLOR_PAIR(1)); // Черный текст на белом фоне
    } else {
        attron(COLOR_PAIR(2)); // Белый текст на черном фоне
    }

    mvprintw(1, 0, "%*s", max_x, ""); // Заполняем всю вторую строку пробелами с цветовой парой
    mvprintw(1, 0, "%-5s %-8s %-5s %-9s %-9s %-5s %-9s %s",
             "PID", "USER", "STATE", "RES_MEM", "VIRT_MEM", "CORES", "START", "COMMAND");

    // Отключаем цветовую пару после второй строки
    if (color_scheme == COLOR_SCHEME_INVERTED) {
        attroff(COLOR_PAIR(1));
    } else {
        attroff(COLOR_PAIR(2));
    }
}

void display_process_info(const ProcessInfo *proc_info) {
    // Обрезаем имя пользователя до 8 символов
    printw("%-5d %-8.8s %-5c %-6.1f MB %-5.1f MB %-5d %-9.8s ",
           proc_info->pid, proc_info->user, proc_info->state,
           proc_info->resident_memory, proc_info->virtual_memory,
           proc_info->cpu_cores, proc_info->start_time + 11);

    // Обрезаем имя команды до символа '/'
    char *slash_pos = strchr(proc_info->command, '/');
```

```
    if (slash_pos) {
        *slash_pos = '\0'; // Устанавливаем символ '/' как конец строки
    }
    printw("%s\n", proc_info->command);
}


void update_display(int start_line, int total_lines, ProcessData *process_data, int process_count,
DisplayMode mode, ColorScheme color_scheme) {
    // Сортировка данных перед отображением
    switch (current_sort) {
        case SORT_BY_PID:
            qsort(process_data, process_count, sizeof(ProcessData), compare_by_pid);
        break;
        case SORT_BY_RESIDENT_MEMORY:
            qsort(process_data, process_count, sizeof(ProcessData), compare_by_resident_memory);
        break;
        case SORT_BY_VIRTUAL_MEMORY:
            qsort(process_data, process_count, sizeof(ProcessData), compare_by_virtual_memory);
        break;
    }

    clear(); // Очистка окна перед новым выводом
    display_header(color_scheme);
    int y = 2; // Текущая строка для вывода информации о процессе/потоке
    int line_count = 2; // Счетчик строк для вывода (учитываем строку заголовка)

    for (int i = 0; i < process_count && line_count < total_lines; i++) {
        if (line_count >= start_line && y < LINES) {
            move(y, 0);
            if (color_scheme == COLOR_SCHEME_INVERTED) {
                attron(COLOR_PAIR(2));
            } else {
                attron(COLOR_PAIR(1));
            }
            display_process_info(&process_data[i].process_info);
            y++;
        }
        line_count++;

        if (mode == SHOW_PROCESSES_AND_THREADS) {
            for (int j = 0; j < process_data[i].thread_count && line_count < total_lines; j++) {
                if (line_count >= start_line && y < LINES) {
                    move(y, 0);
                    if (color_scheme == COLOR_SCHEME_INVERTED) {
                        attron(COLOR_PAIR(2));
                    } else {
                        attron(COLOR_PAIR(1));
                    }
                    display_thread_info(&process_data[i].threads[j]);
                    y++;
                }
                line_count++;
            }
        }
    }
    refresh();
}

void display_help(ColorScheme color_scheme) {
    clear(); // Очищаем окно перед выводом справки

    // Определяем символы для заполнения пространства вокруг текста справки
    char fill_char = ' ';
    if (color_scheme == COLOR_SCHEME_INVERTED) {
        fill_char = '#'; // Для инвертированной схемы используем пробел
    } else {
        fill_char = '#'; // Для стандартной схемы используем решетку
    }

    // Определяем цвет заполнения в зависимости от выбранной темы
    int fill_color_pair = 1;
    if (color_scheme == COLOR_SCHEME_INVERTED) {
        fill_color_pair = 2; // В светлой теме цвет черный
    }

    // Получаем размеры окна
    int max_y, max_x;
```

```
    getmaxyx(stdscr, max_y, max_x);

    // Определяем координаты вывода для каждой строки
    int y_center = (max_y / 2)-2; // Центральная координата по вертикали
    int x_cetner = max_x / 2; // Центральная координата по горизонтали

    // Вывод справки
    mvprintw(y_center - 3, x_cetner - 6, "=== Help ===");
    mvprintw(y_center - 1, x_cetner - 20, "Press 'p' to sort by PID (process ID)");
    mvprintw(y_center, x_cetner - 20, "Press 'r' to sort by resident memory");
    mvprintw(y_center + 1, x_cetner - 20, "Press 'v' to sort by virtual memory");
    mvprintw(y_center + 2, x_cetner - 20, "Press 'k' to kill a process or thread");
    mvprintw(y_center + 3, x_cetner - 20, "Press 'q' to quit");

    // Добавленные строки
    mvprintw(y_center + 4, x_cetner - 20, "Press 't' to show threads");
    mvprintw(y_center + 5, x_cetner - 20, "Press 'key down' to go 20 lines down");
    mvprintw(y_center + 6, x_cetner - 20, "Press 'key up' to go 20 lines up");
    mvprintw(y_center + 7, x_cetner - 20, "Press 'z' to change color scheme");

    // Пустая строка
    mvprintw(y_center + 8, x_cetner - 20, "");

    // Заполняем пространство вокруг текста справки
    attron(COLOR_PAIR(fill_color_pair)); // Устанавливаем цвет для заполнения
    for (int i = 0; i < max_y; i++) {
        for (int j = 0; j < max_x; j++) {
            if (i < y_center - 4 || i > y_center + 8 || j < x_cetner - 30 || j > x_cetner + 30) {
                mvaddch(i, j, fill_char);
            }
        }
    }
    attroff(COLOR_PAIR(fill_color_pair)); // Отключаем цвет заполнения

    refresh();

    // Ожидаем нажатия любой клавиши или истечения времени
    timeout(30000); // Ожидание 30 секунд
    getch(); // Ожидаем нажатия клавиши
    clear(); // Очищаем экран после завершения отображения справки
    refresh();
}
```

Файл display.h

```
#ifndef DISPLAY_H
#define DISPLAY_H

#include "display.h"
#include "processes.h"

#define MAX_THREADS_PER_PROCESS 100  // Максимальное количество потоков на процесс

typedef enum {
    SHOW_PROCESSES,
    SHOW_PROCESSES_AND_THREADS
} DisplayMode;

typedef enum {
    COLOR_SCHEME_DEFAULT,
    COLOR_SCHEME_INVERTED
} ColorScheme;

void display_help(ColorScheme color_scheme);
void display_thread_info();
void display_header(ColorScheme color_scheme);
void display_time(ColorScheme color_scheme);
void display_process_info(const ProcessInfo *proc_info);
void update_display(int start_line, int total_lines, ProcessData *process_data, int process_count,
DisplayMode mode, ColorScheme color_scheme);
void kill_process_or_thread();
#endif /* DISPLAY_H */
```

Файл main.c

```
#include <curses.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <dirent.h>
#include <string.h>
#include "processes.h"
#include "display.h"
#include "threads.h"
#include "control.h"
#include "read.h"

#define MAX_PROCESSES 1000  // Максимальное количество процессов для отображения
#define REFRESH_INTERVAL 1000  // Интервал обновления в миллисекундах
#define SCROLL_LINES 20  // Количество строк для прокрутки по стрелкам

int main() {
    initscr();          // Инициализация ncurses
    start_color();      // Включение поддержки цвета
    noecho();           // Отключение эха вводимых символов
    cbreak();           // Включение режима cbreak
    keypad(stdscr, TRUE); // Включение поддержки клавиш
    mousemask(ALL_MOUSE_EVENTS, NULL); // Включение обработки всех событий мыши

    // Инициализация цветовых пар
    init_pair(1, COLOR_WHITE, COLOR_BLACK); // Стандартная цветовая схема (белый текст на черном
фоне)
    init_pair(2, COLOR_BLACK, COLOR_WHITE); // Инвертированная цветовая схема (черный текст на белом
фоне)

    int start_line = 0;  // Первая видимая строка
    int total_lines = 0; // Общее количество строк для отображения
    DisplayMode mode = SHOW_PROCESSES; // Режим отображения по умолчанию
    ColorScheme color_scheme = COLOR_SCHEME_DEFAULT; // Цветовая схема по умолчанию

    while (1) {
        DIR *dir;
        struct dirent *entry;

        dir = opendir("/proc");
        if (!dir) {
            perror("opendir(/proc)");
            endwin();
            return 1;
        }

        ProcessData process_data[MAX_PROCESSES]; // Сохранение информации о процессах и потоках
        int process_count = 0; // Обнуляем количество процессов
        total_lines = 1; // Обнуляем количество строк (учитываем строку заголовка)

        while ((entry = readdir(dir)) != NULL) {
            if (entry->d_type == DT_DIR) {  // Проверяем тип элемента и его идентификатор
                int pid = atoi(entry->d_name);  // Преобразуем имя каталога в целочисленный PID
                if (pid > 0 && process_count < MAX_PROCESSES) { // Проверяем, что PID положительный
и количество процессов еще не достигло максимума
                    ProcessInfo proc_info;
                    get_process_info(&proc_info, pid); // Получаем информацию о процессе по его PID
                    process_data[process_count].process_info = proc_info; // Сохраняем информацию о
процессе в массиве process_data
                    process_data[process_count].thread_count = 0; // Обнуляем счетчик потоков для
данного процесса
                    total_lines++; // Увеличиваем общее количество строк для отображения

                    if (mode == SHOW_PROCESSES_AND_THREADS) {  // Если режим отображения включает
информацию о потоках, получаем ее
                        char path[256];
                        DIR *task_dir;
                        struct dirent *task_entry;
                        snprintf(path, sizeof(path), "/proc/%d/task", proc_info.pid); // Формируем
путь к каталогу с потоками процесса
                        task_dir = opendir(path); // Открываем каталог
                        if (task_dir) {
                            while ((task_entry = readdir(task_dir)) != NULL) {  // Перебираем все
элементы в каталоге потоков
                                // Проверяем тип элемента и его имя
```

```
                                if (task_entry->d_type == DT_DIR && task_entry->d_name[0] !=
                                    '.' && process_data[process_count].thread_count <
MAX_THREADS_PER_PROCESS) {
                                    int tid = atoi(task_entry->d_name);  // Преобразуем имя каталога
в целочисленный TID
                                    ThreadInfo thread_info;
                                    get_thread_info(&thread_info, proc_info.pid, tid);  // Получаем
информацию о потоке по его PID и TID

process_data[process_count].threads[process_data[process_count].thread_count] = thread_info;
                                    process_data[process_count].thread_count++; // Увеличиваем
счетчик потоков
                                    total_lines++; // Увеличиваем общее количество строк для
отображения
                                }
                            }
                            closedir(task_dir);
                        }
                    }
                    process_count++;
                }
            }
        }
        closedir(dir);
        update_display(start_line, total_lines, process_data, process_count, mode, color_scheme);

        timeout(REFRESH_INTERVAL);  // Устанавливаем таймаут ожидания ввода
        int ch = getch();
        if (ch == KEY_MOUSE) {
            MEVENT event;
            if (getmouse(&event) == OK) {
                if (event.bstate & BUTTON4_PRESSED) { // Прокрутка вверх
                    if (start_line > 0) start_line--;
                } else if (event.bstate & BUTTON5_PRESSED) { // Прокрутка вниз
                    // Увеличиваем начальную строку, если это возможно
                    if (start_line < total_lines - (LINES - 1)) start_line++;
                }
                update_display(start_line, total_lines, process_data, process_count, mode,
color_scheme);
            }
        } else if (ch == KEY_UP) {
            if (start_line > SCROLL_LINES) {
                start_line -= SCROLL_LINES;
            } else {
                start_line = 0; // Сбрасываем начало строки при переключении режима
            }
            update_display(start_line, total_lines, process_data, process_count, mode,
color_scheme);
        } else if (ch == KEY_DOWN) {
            if (start_line + SCROLL_LINES < total_lines - (LINES - 1)) {
                start_line += SCROLL_LINES;
            } else {
                start_line = total_lines - (LINES - 1);
                if (start_line < 0) start_line = 0;
            }
            update_display(start_line, total_lines, process_data, process_count, mode,
color_scheme);
        } else if (ch == 't') {
            if (mode == SHOW_PROCESSES) {
                mode = SHOW_PROCESSES_AND_THREADS;
            } else {
                mode = SHOW_PROCESSES;
            }
            start_line = 0;  // Сбрасываем начало строки при переключении режима
            update_display(start_line, total_lines, process_data, process_count, mode,
color_scheme);
        } else if (ch == 'h' || ch == 'H') {
            display_help(color_scheme);
        }else if (ch == 'z' || ch == 'Z') {
            if (color_scheme == COLOR_SCHEME_DEFAULT) {
                // Включение инвертированной цветовой схемы для всего окна
                wbkgd(stdscr, COLOR_PAIR(2));
                color_scheme = COLOR_SCHEME_INVERTED;
            } else {
                // Включение стандартной цветовой схемы для всего окна
                wbkgd(stdscr, COLOR_PAIR(1));
                color_scheme = COLOR_SCHEME_DEFAULT;
            }
```

```
            start_line = 0;  // Сбрасываем начало строки при переключении цветовой схемы
            update_display(start_line, total_lines, process_data, process_count, mode,
color_scheme);
        } else if (ch == 'q' || ch == 'Q') {
            break; // Завершение программы при нажатии 'q'
        } else if (ch == ERR) {
            // Таймаут достигнут, обновляем экран
            update_display(start_line, total_lines, process_data, process_count, mode,
color_scheme);
        }
        handle_user_input(ch);
        update_display(start_line, total_lines, process_data, process_count, mode, color_scheme);
    }

    endwin(); // Завершение работы ncurses
    return 0;
}
```

Файл processes.h

```
#ifndef PROCESSES_H
#define PROCESSES_H
#include "threads.h"

#define MAX_THREADS_PER_PROCESS 100

typedef struct {
    int pid;                  // ID процесса
    char user[50];            // Имя пользователя вызвавшего процесс
    char state;               // Состояние процесса
    double resident_memory;   // Потребление физической памяти (MB)
    double virtual_memory;    // Виртуальная память, используемая процессом (MB)
    int cpu_cores;            // Количество ядер процессора, используемых процессом
    int threads;              // Количество потоков
    char start_time[20];      // Дата и время запуска
    char command[100];        // Имя команды, запустившей процесс
} ProcessInfo;

typedef struct {
    ProcessInfo process_info;                    // Информация о процессе
    ThreadInfo threads[MAX_THREADS_PER_PROCESS]; // Информация о потоках процесса
    int thread_count;                            // Количество потоков процесса
} ProcessData;                                   // Структура для хранения информации о процессе и
его потоках

int compare_by_pid(const void *a, const void *b);
int compare_by_resident_memory(const void *a, const void *b);
int compare_by_virtual_memory(const void *a, const void *b);

#endif // PROCESSES_H
```

Файл read.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pwd.h>
#include <time.h>
#include "processes.h"
#include "threads.h"

void get_thread_info(ThreadInfo *thread_info, int pid, int tid) {
    char path[256];
    FILE *file;
    thread_info->tid = tid;

    snprintf(path, sizeof(path), "/proc/%d/task/%d/comm", pid, tid);
    file = fopen(path, "r");
    if (file) {
        if (fgets(thread_info->name, sizeof(thread_info->name), file)) {
            thread_info->name[strcspn(thread_info->name, "\n")] = '\0';
        }
        fclose(file);
    }
```

```
    snprintf(path, sizeof(path), "/proc/%d/task/%d/status", pid, tid);
    file = fopen(path, "r");
    if (file) {
        char buffer[256];
        while (fgets(buffer, sizeof(buffer), file)) {
            if (strncmp("State:", buffer, 6) == 0) {
                sscanf(buffer, "State: %c", &thread_info->state);
                break;
            }
        }
        fclose(file);
    }
}

// Функция для подсчета количества ядер
int count_cpu_cores(const char *cpu_list) {
    int cores = 1;
    for (const char *p = cpu_list; *p; p++) {
        if (*p == ',') {
            cores++;
        }
    }
    return cores;
}

// Функция для чтения времени запуска системы
time_t get_system_uptime() {
    FILE *file = fopen("/proc/uptime", "r");
    if (!file) {
        perror("Failed to open /proc/uptime");
        return 0;
    }
    double uptime;
    fscanf(file, "%lf", &uptime);
    fclose(file);
    return (time_t)uptime;
}

// Функция для чтения информации о процессе
void get_process_info(ProcessInfo *proc_info, int pid) {
    char path[256];
    char buffer[256];
    FILE *file;
    struct passwd *pw;

    // Заполняем PID
    proc_info->pid = pid;

    // Чтение информации из /proc/[pid]/status для получения имени пользователя, состояния процесса,
виртуальной памяти и команды
    snprintf(path, sizeof(path), "/proc/%d/status", pid);
    file = fopen(path, "r");
    if (file) {
        while (fgets(buffer, sizeof(buffer), file)) {
            if (strncmp(buffer, "Uid:", 4) == 0) {
                int uid;
                sscanf(buffer, "Uid: %d", &uid);
                pw = getpwuid(uid);
                if (pw) {
                    strncpy(proc_info->user, pw->pw_name, sizeof(proc_info->user) - 1);
                }
            } else if (strncmp(buffer, "VmSize:", 7) == 0) {
                unsigned long vm_size;
                sscanf(buffer, "VmSize: %lu kB", &vm_size);
                proc_info->virtual_memory = vm_size / 1024.0 / 1024.0; // Конвертируем в MB
            } else if (strncmp(buffer, "State:", 6) == 0) {
                sscanf(buffer, "State: %c", &proc_info->state);
            } else if (strncmp(buffer, "Name:", 5) == 0) {
                sscanf(buffer, "Name: %s", proc_info->command);
            } else if (strncmp(buffer, "Cpus_allowed_list:", 18) == 0) {
                // Здесь считываем информацию о привязке к ядрам
                char cpu_list[256];
                sscanf(buffer, "Cpus_allowed_list: %s", cpu_list);
                proc_info->cpu_cores = count_cpu_cores(cpu_list); // Подсчитываем количество ядер
            } else if (strncmp(buffer, "Threads:", 8) == 0) {
                sscanf(buffer, "Threads: %d", &proc_info->threads);
            }
        }
```

```
        fclose(file);
    }

    // Чтение информации из /proc/[pid]/stat для получения потребления физической памяти и времени
запуска
    snprintf(path, sizeof(path), "/proc/%d/stat", pid);
    file = fopen(path, "r");
    if (file) {
        long rss;
        unsigned long utime, stime, starttime;
        fscanf(file, "%*d %*s %*c %*d %*d %*d %*d %*d %*d %*u %*u %*u %*u %lu %lu %*d %*d %*d %*d
%llu", &utime, &stime, &starttime);
        fclose(file);

        // Получаем текущее время в секундах
        time_t now = time(NULL);
        // Время работы системы в секундах
        time_t uptime = get_system_uptime();
        // Время запуска процесса (текущее время - (время работы системы - время старта процесса))
        time_t start_time = now - (uptime - (starttime / sysconf(_SC_CLK_TCK)));

        struct tm start_tm;
        localtime_r(&start_time, &start_tm);
        strftime(proc_info->start_time, sizeof(proc_info->start_time), "%Y-%m-%d %H:%M:%S",
&start_tm);

        snprintf(path, sizeof(path), "/proc/%d/statm", pid);
        file = fopen(path, "r");
        if (file) {
            fscanf(file, "%*d %ld", &rss);
            fclose(file);
            proc_info->resident_memory = rss * (sysconf(_SC_PAGESIZE) / 1024.0 / 1024.0); //
Конвертируем в MB
        }
    }
}
```

Файл read.h

```
#ifndef READ_H
#define READ_H

#include "processes.h"
#include "threads.h"


void get_thread_info(ThreadInfo *thread_info, int pid, int tid);
void get_process_info(ProcessInfo *proc_info, int pid);
#endif // READ_H
```

Файл threads.h

```
#ifndef THREADS_H
#define THREADS_H

typedef struct {
    int tid;         // ID потока
    char state;      // Сстояние потока
    char name[16];   // Имя потока
} ThreadInfo;

#endif //THREADS_H
```

ПРИЛОЖЕНИЕ Д
(Обязательное)

Ведомость документов