Министерство образования и науки РФ

Федеральное государственное автономное

образовательное учреждение высшего образования

«Национальный исследовательский университет ИТМО»

**факультет программной инженерии и компьютерной техники**

**ЛАБОРАТОРНАЯ РАБОТА №1**

по дисциплине

‘Операционные системы’

*Выполнил:*

Студент группы P33312

Соболев Иван Александрович

*Преподаватель:*

Пашнин Александр Денисович

Санкт-Петербург, 2023

Оглавление

[Задание: 3](#_Toc148095278)

[Выполнение: 3](#_Toc148095279)

[CPU: 3](#_Toc148095280)

[Cache: 3](#_Toc148095281)

[IO: 3](#_Toc148095282)

[Memory 3](#_Toc148095283)

[Network 3](#_Toc148095284)

[Pipe 3](#_Toc148095285)

[Sched 3](#_Toc148095286)

[Выводы по лабораторной работе: 3](#_Toc148095287)

[В ходе выполнения лабораторной работы я применил на практике утилиты для мониторинга процессов и системного анализа. 3](#_Toc148095288)

# Задание:

Изображение выглядит как текст, Шрифт, снимок экрана, информация

Автоматически созданное описание Изображение выглядит как текст, снимок экрана, Шрифт, алгебра

Автоматически созданное описание

**Исходный код всех скриптов:** [**https://github.com/Ivanio1/itmo-os/tree/main/lab1**](https://github.com/Ivanio1/itmo-os/tree/main/lab1)

****

# Выполнение:

Вариант:

cpu: [int128decimal128, decimal64];

cache: [cache-ways, l1cache];

io: [iomix, ioprio];

memory: [lockbus, fork-vm];

network: [sockdiag, netlink-proc];

pipe: [pipe-size, pipeherd-yield];

sched: [sched-runtime, sched-prio]

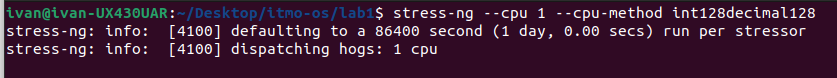
## CPU:

**1) Запустим stress-ng с первым параметром.**

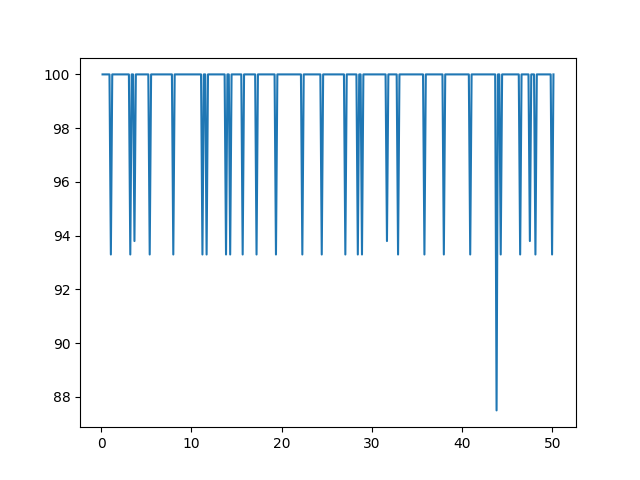
**--int128decimal128** 1000 итераций сочетания 128-разрядных целочисленных и 128-разрядных десятичных операций с плавающей запятой.

Команда запуска: stress-ng --cpu 1 --cpu-method int128decimal128

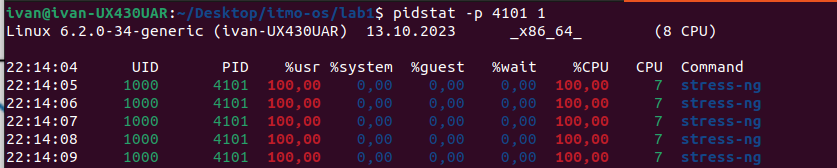
Запуск:



Построим график с помощью питоновского скрипта (top):

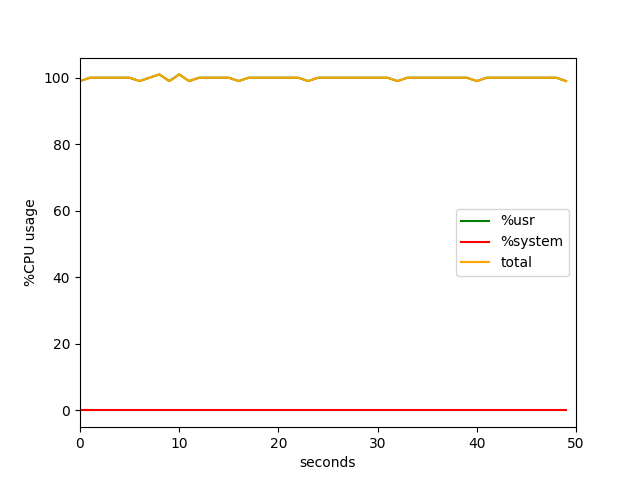


Также посмотрим на потребление CPU с помощью команды pidstat:

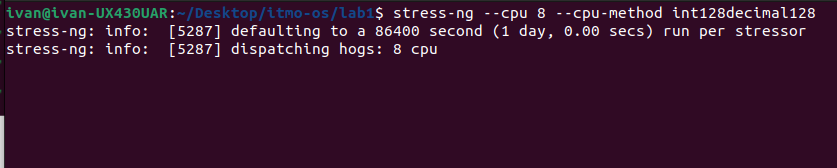
Данная команда выводит статистику по потоку -p PID раз в секунду.

Как мы можем видеть 1 ядро загружено на 100%.

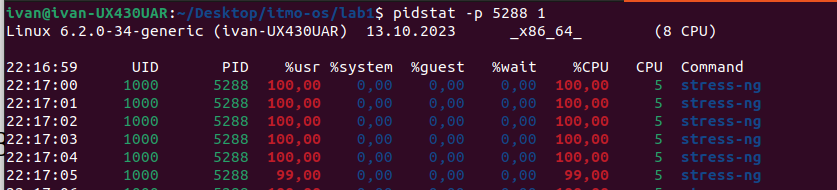
График (pidstat):

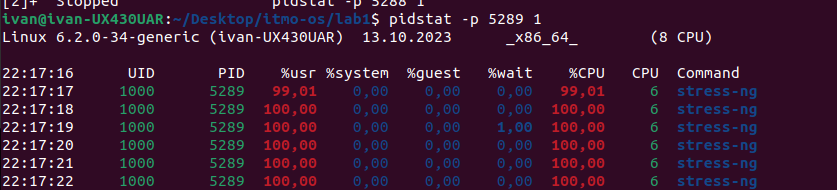


Попробуем запустить программу на 8 процессорных ядрах. (В системе 8 ядер)



Имеем похожую картину:

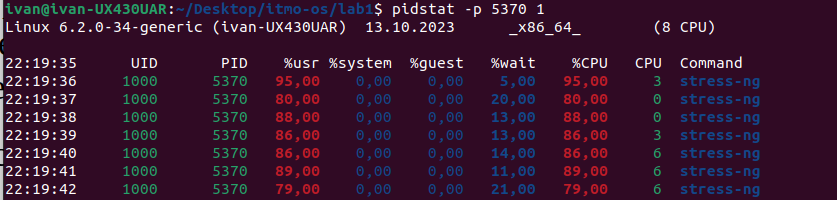




Каждое из 8 ядер нагружается практически на 100% своим тестом. Можно увидеть на каком ядре работает процесс, обратив внимание на параметр CPU в выводе pidstat.

Посмотрим, что будет при запуске 9 тестов:

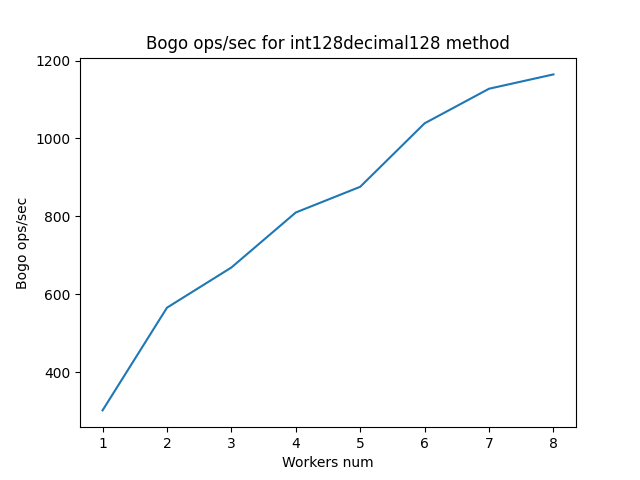




Теперь же ситуация другая. Так как процессов больше, чем ядер, то возникает конкуренция за ресурсы. Она на многоядерных процессорах возникает, когда более одного процесса или потока пытаются использовать общие аппаратные ресурсы, такие как ядра процессора, кэш-память, оперативная память (RAM) и другие системные ресурсы. Эта конкуренция может влиять на производительность и загрузку процессора.

Чтобы добиться максимальной загрузки системы, нужно запускать n тестов на n-ядерном процессоре, тогда каждое ядро будет использоваться на 100%.

Посмотрим также на график bogoops:

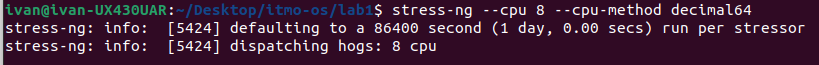
Количество bogo операций увеличивается с увеличением количества тестов.

**2) Посмотрим второй параметр**

**-- decimal64** 1000 итераций сочетания 64-разрядных десятичных операций с плавающей запятой

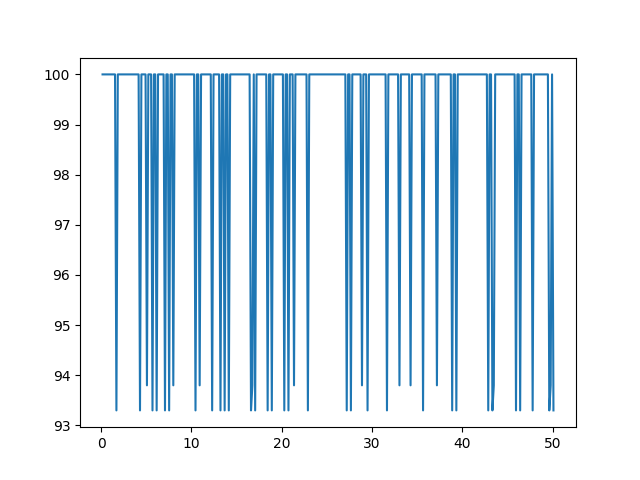
Команда запуска: stress-ng --cpu 8 --cpu-method decimal64

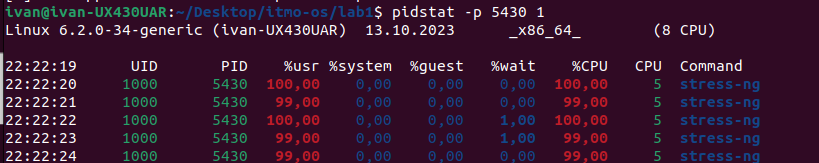
Запуск:

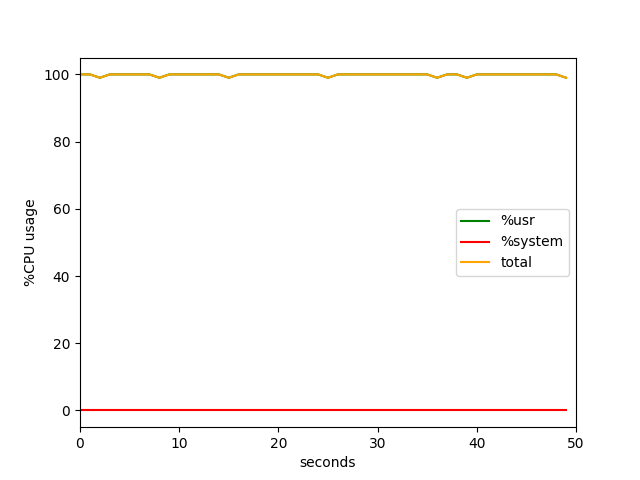


Запустим сразу на 8 ядрах.

График загрузки одного из ядер (top).

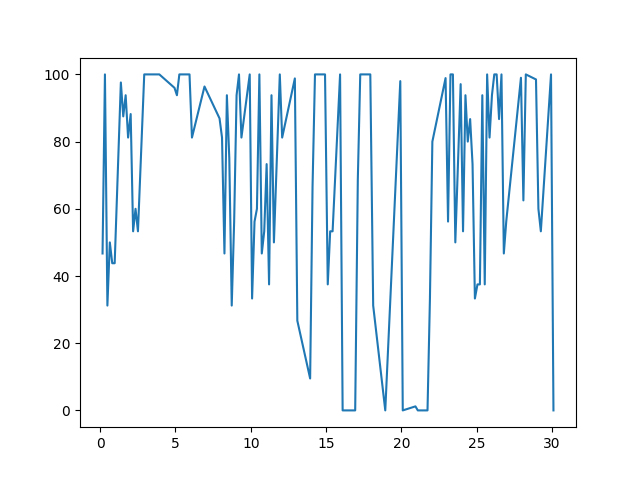


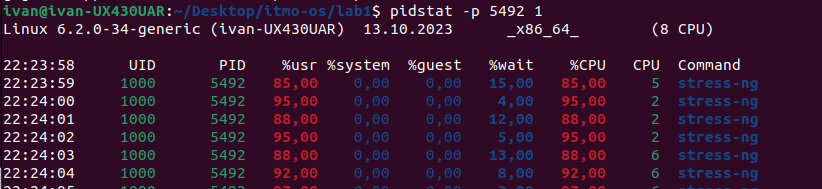
График (pidstat):



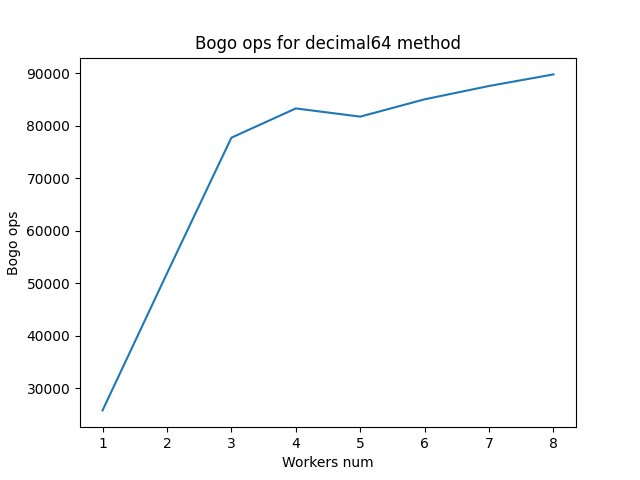
Запустим 9 тестов:





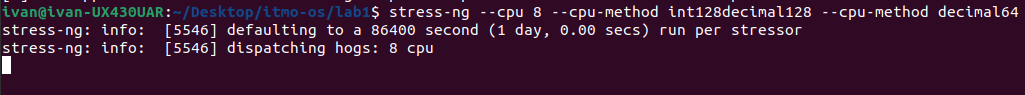
Имеем аналогичную ситуацию. Это ожидалось, так как мы просто изменили метод нагрузки.

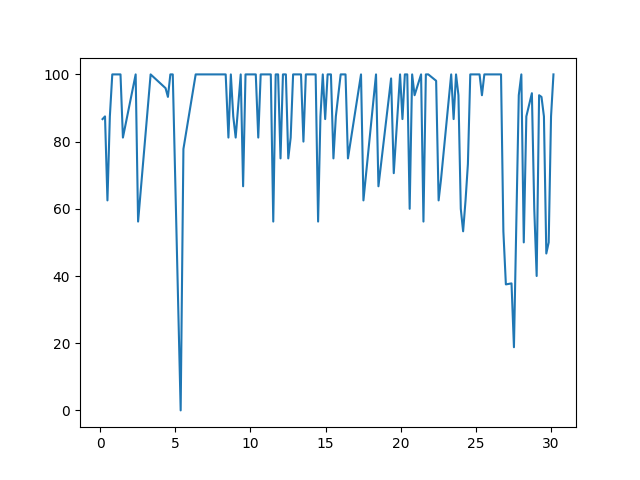
Посмотрим также на график bogoops:

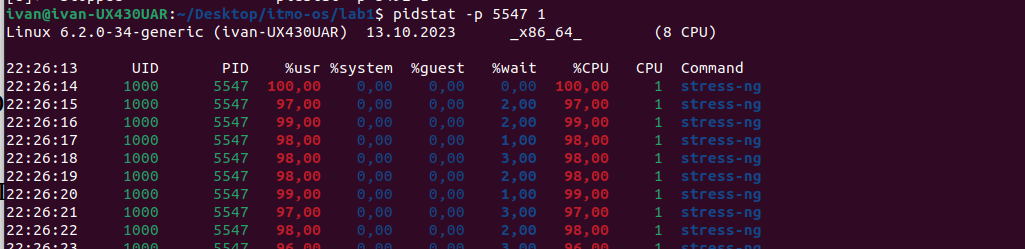


Количество bogo операций увеличивается с увеличением количества тестов.

**3) Запустим тесты сразу с двумя параметрами.**

****





Можем заметить, что при запуске двух методов нагрузки мы видим некие просадки в производительности. Процессор вынужден работать более интенсивно, что может вызвать просадки из-за следующих факторов:

* Увеличение общей нагрузки: Запуск двух методов увеличивает общую нагрузку на процессор, что может создавать более сильные просадки.
* Конфликт ресурсов: Разные методы могут конфликтовать за доступ к вычислительным ресурсам (Например, оперативная память).
* Увеличение конкуренции: Запуск двух методов может вызвать конкуренцию между ними, что может привести к борьбе за ресурсы процессора и, следовательно, к просадкам в процентной загрузке.

**Выводы по мониторингу CPU:** для максимальной производительности n-ядерного процессора, необходимо запускать нагрузочные тесты ровно на n ядер. При увеличении данной цифры производительность падает. Также при увеличении числа методов нагрузочного тестирования производительность также снижается.

Утилита top и pidstat показали примерно одинаковые цифры.

## Cache:

1. **Первый параметр**

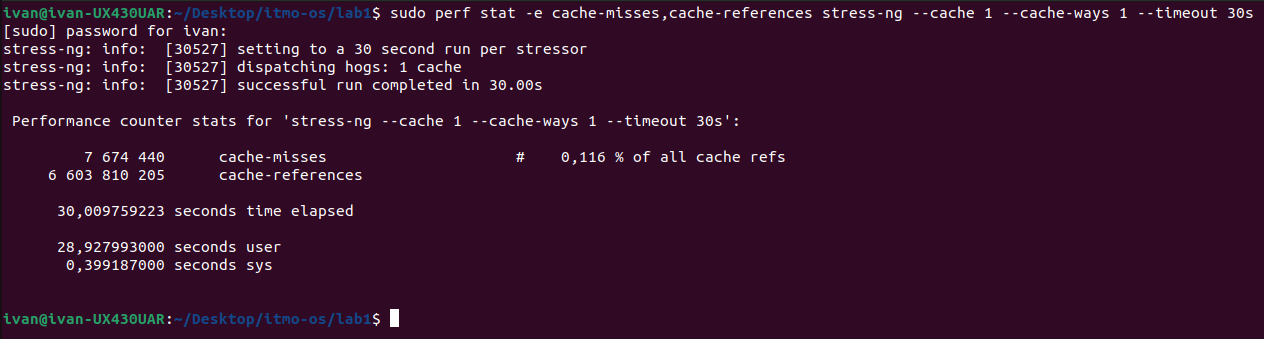
**--cache-ways N** указывает количество способов кэширования для тестирования.

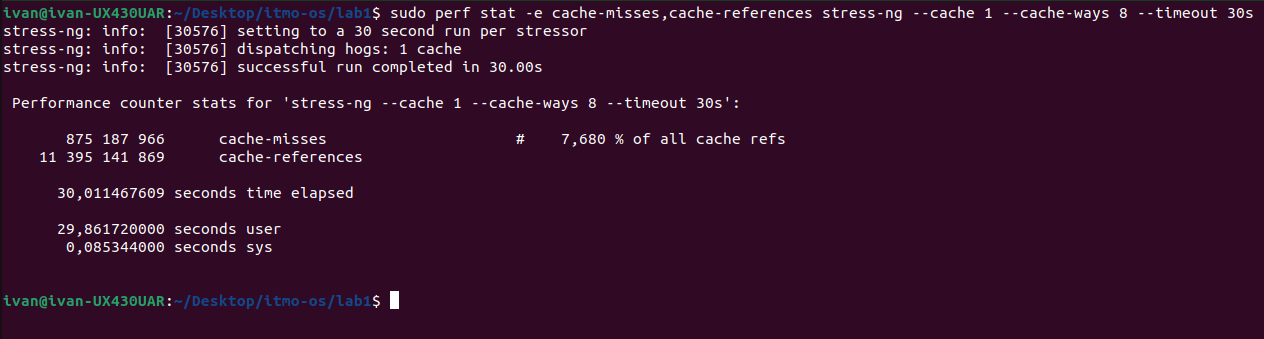
Команда запуска: sudo perf stat -e cache-misses,cache-references stress-ng --cache 1 --cache-ways 1 --timeout 30s

Кэш-память в современных процессорах обычно имеет несколько уровней, такие как L1, L2 и L3. Каждый уровень кэша может иметь определенное количество способов ассоциации, что влияет на то, сколько блоков данных может храниться в кэше.

Параметр "cache-ways" в stress-ng позволяет определить, сколько способов ассоциации должно использоваться в кэше.

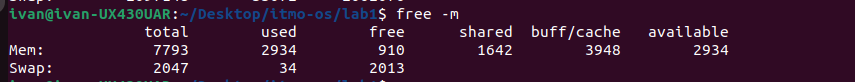
С помощью данной команды мы запустим один тестер кэша с одним способом ассоциации, далее с помощью утилиты perf соберем информацию о кэш-промахах и обращениях к кэш памяти за 30 секунд работы тестера.

Увеличим количества способов ассоциации:

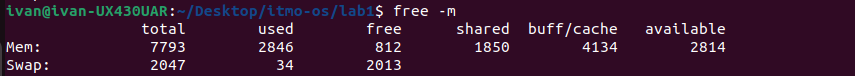
Статистика по обращениям и промахам увеличилась. Ожидаемо, ведь мы заставляем систему обрабатывать различные конфигурации кэша.

Также проведем мониторинг утилитой free:

До запуска стресс-теста:



После запуска стресс-теста:

Значение памяти, которая ииспользуется буферами ядра и кэшем увеличилось.

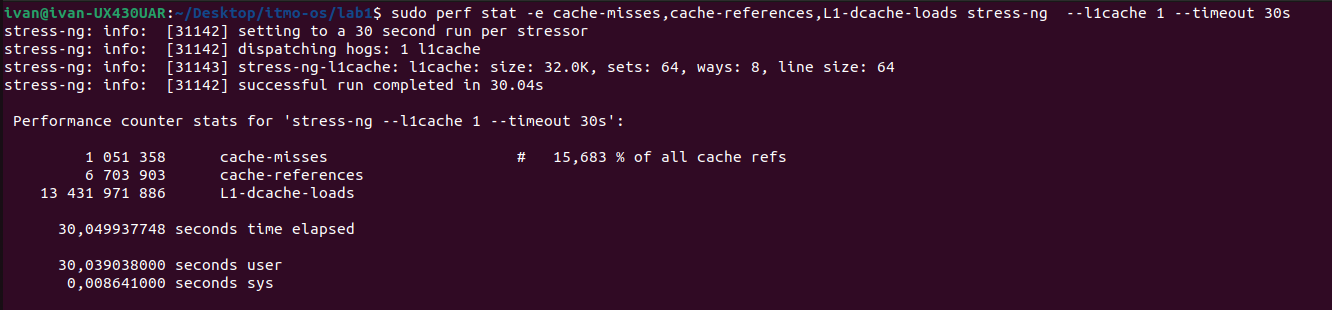
1. **Второй параметр**

**-- l1cache N** запускает N рабочих процессов, которые используют кэш ЦП уровня 1 при чтении и записи.

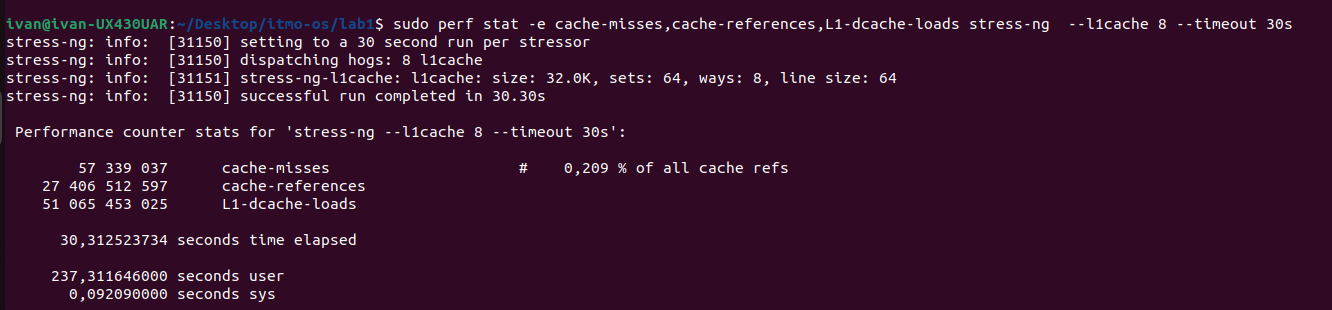
Команда запуска: sudo perf stat -e cache-misses,cache-references,L1-dcache-loads stress-ng --l1cache 1 --timeout 30s

L1-dcache-loads – статистика по загрузке кэша первого уровня.

Статистика утилитой perf:



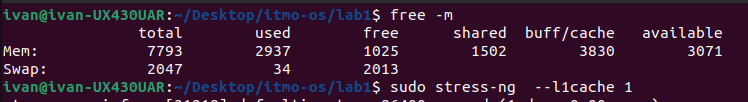
Увеличим количество рабочих процессов:



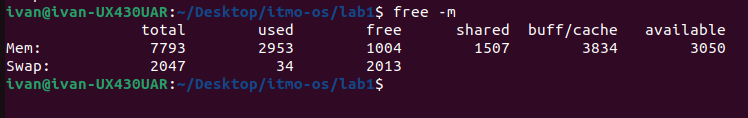
Загрузка кэша увеличивается.

Также проведем мониторинг утилитой free:

До запуска стресс-теста:

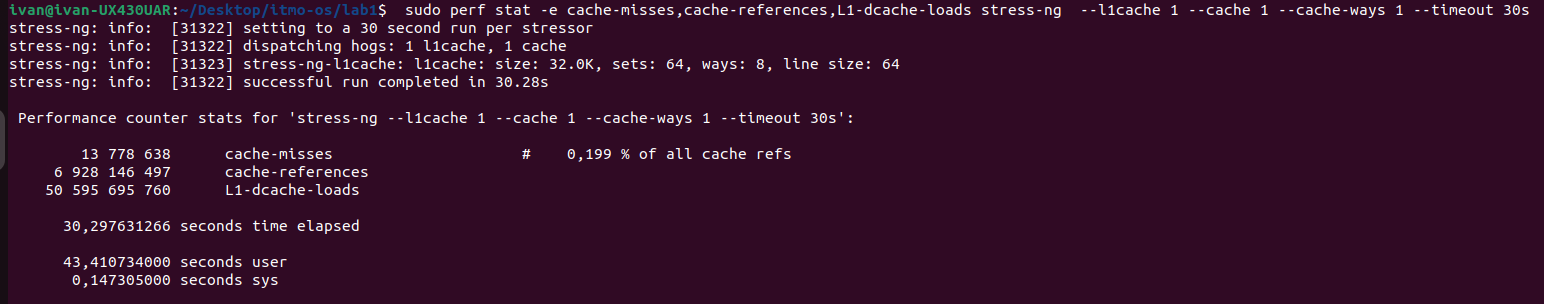


После запуска стресс-теста:

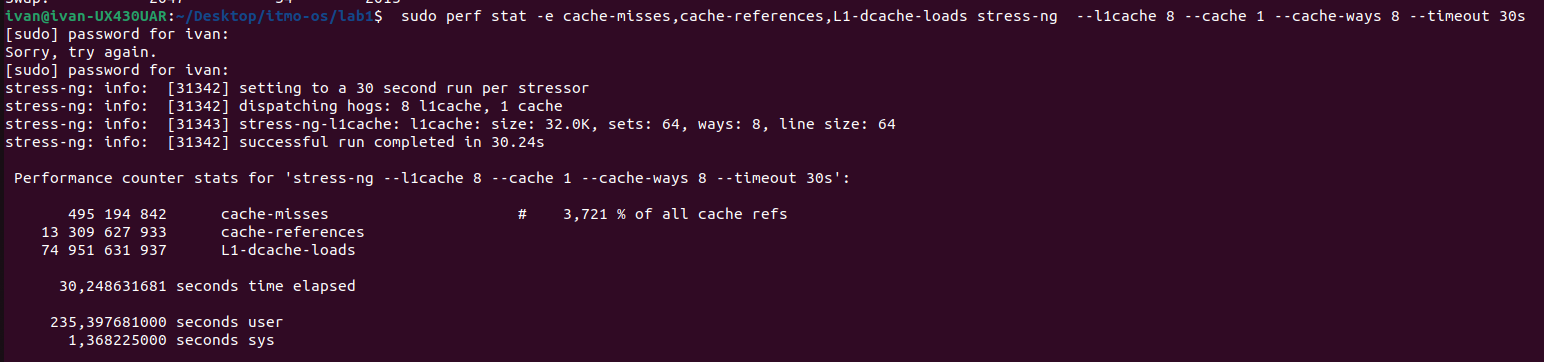


1. **Запуск с двумя параметрами**

Команда запуска: sudo perf stat -e cache-misses,cache-references,L1-dcache-loads stress-ng --l1cache 1 --cache 1 --cache-ways 1 --timeout 30s



Увеличим нагрузку:



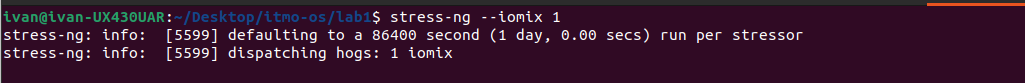
**Выводы по мониторингу кэша:** Был проведен мониторинг использования и нагрузки кэша с помощью утилит perf и free. По окончанию мониторинга можно сделать вывод, что чем больше процессов работает с кэшом, тем больше увеличивается на него нагрузка, увеличивается число кэш-промахов.

## IO:

1. **Первый параметр**

**-- iomix N** запускает N рабочих процессов, которые выполняют сочетание последовательных, случайных и отображаемых в память операций чтения/записи. Создается несколько дочерних процессов, которые совместно используют один файл и выполняют различные операции ввода-вывода с одним и тем же файлом.

Команда запуска: stress-ng --iomix 1

Запустим iomix с аргументом 1, так как будут создаваться еще дочерние процессы.

Посмотрим на питоновский график процента нагрузки подсистемы ввода-вывода.

Внутри питоновского скрипта используется команда iotop с флагами "-P" "-b" "-n" "-d" и считывается процент загрузки io.

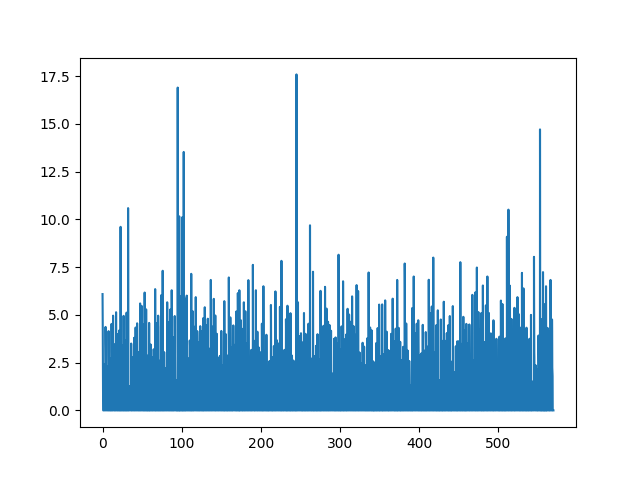
Флаги:

-P: Этот флаг используется для определения фильтрации вывода iotop по процессам.

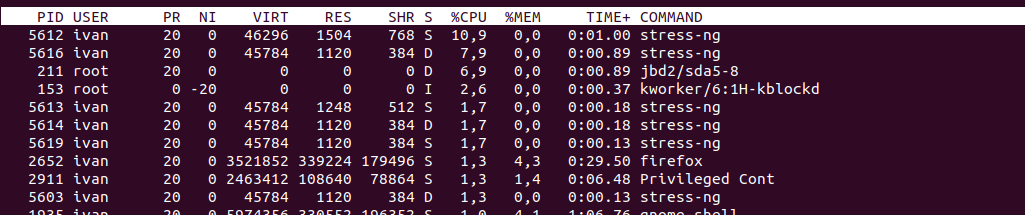
-b: Этот флаг указывает iotop на использование режима "батч-режима". В этом режиме iotop не выводит интерактивную таблицу, а вместо этого она выводит обновления активности ввода-вывода.

-n: Этот флаг определяет количество итераций.

-d: Этот флаг устанавливает интервал между обновлениями активности ввода-вывода в секундах.

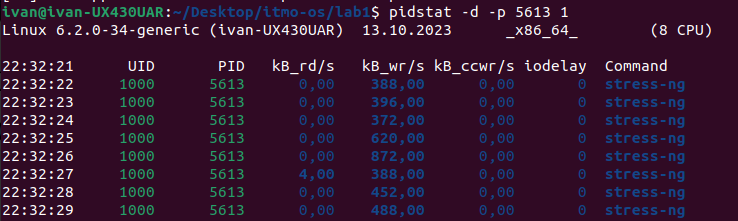


Посмотрим на процессы (top):



Можем увидеть, что и в правду плодятся дополнительные дочерние процессы.

Также посмотрим на один процесс, с помощью команды pidstat:

Можем видеть, что чтение/запись идут, но подсистема полностью не загружена. Попробуем увеличить количество рабочих процессов и построим график:

Запущенная команда stress-ng --iomix 20

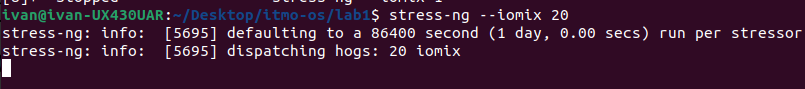
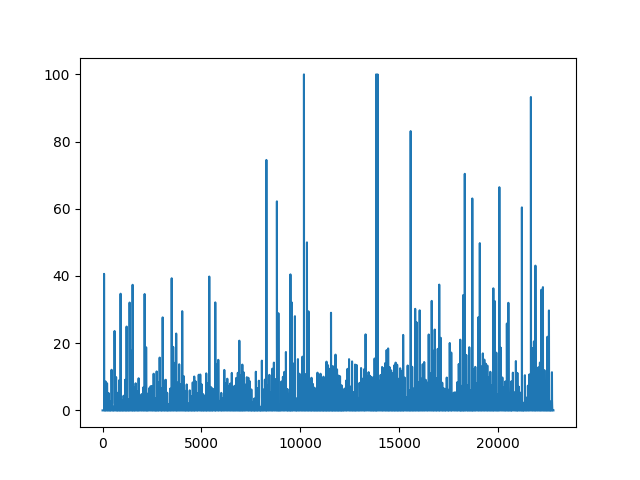


График:



Ожидаемо, что с увеличением процессов работы с подсистемой увеличивается и ее загрузка.

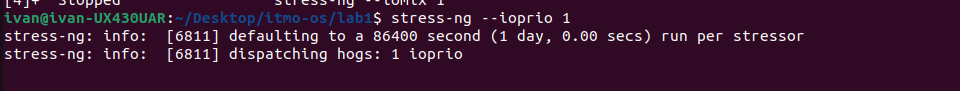
1. **Второй параметр**

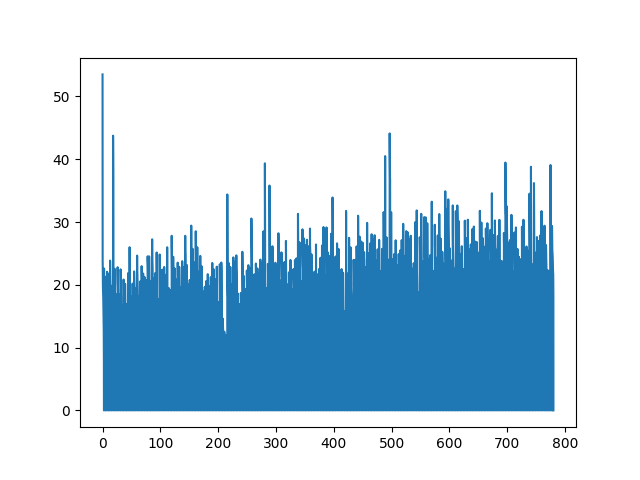
**-- ioprio N** запусткает N рабочих процессов, которые выполняют системные вызовы ioprio\_get(2) и ioprio\_set(2)

Системный вызов ioprio\_get(2) позволяет процессу получить текущий приоритет ввода-вывода (I/O) для указанного файлового дескриптора.

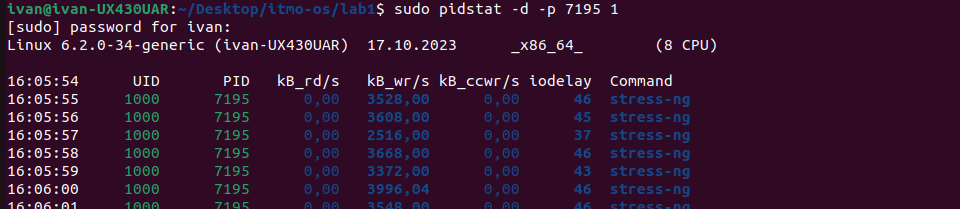
Системный вызов ioprio\_set(2) используется для установки приоритета ввода-вывода для указанного файлового дескриптора.

Команда запуска: sudo stress-ng —ioprio 1



График:

Посмотрим на процесс с помощью pidstat:



Увеличим количество рабочих процессов:

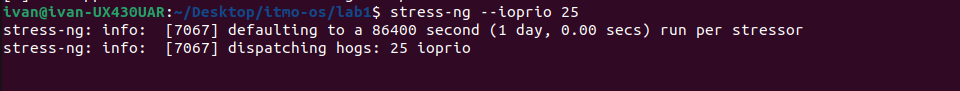
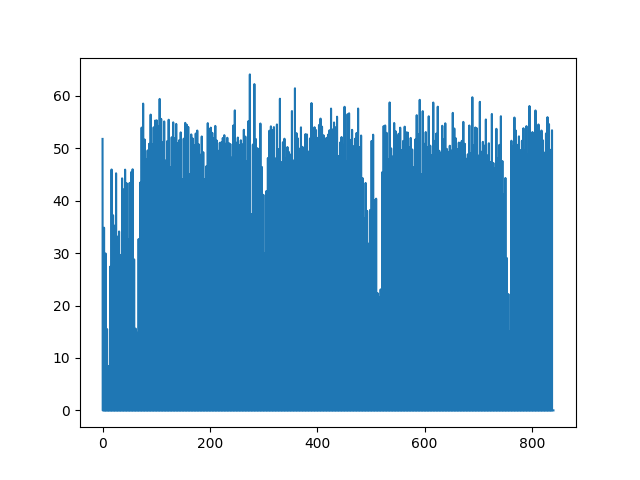


График:



Ожидаемо, что с увеличением процессов работы с подсистемой увеличивается и ее загрузка.

1. **Запуск с двумя параметрами**

Команда запуска: stress-ng --iomix 1 --ioprio 1

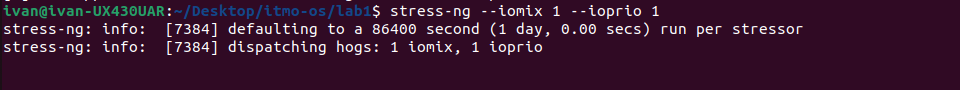
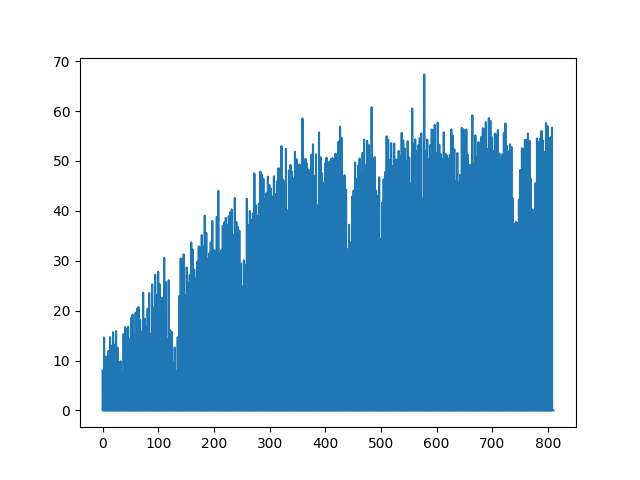


График:



**Выводы по мониторингу подсистемы ввода-вывода:** Был произведен мониторинг системы ввода/вывода утилитами iotop и pidstat. C увеличением процессов работы с подсистемой увеличивается и ее загрузка.

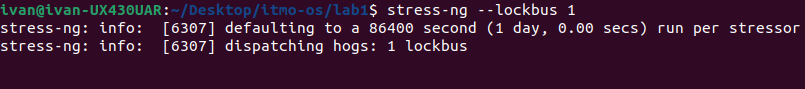
## Memory

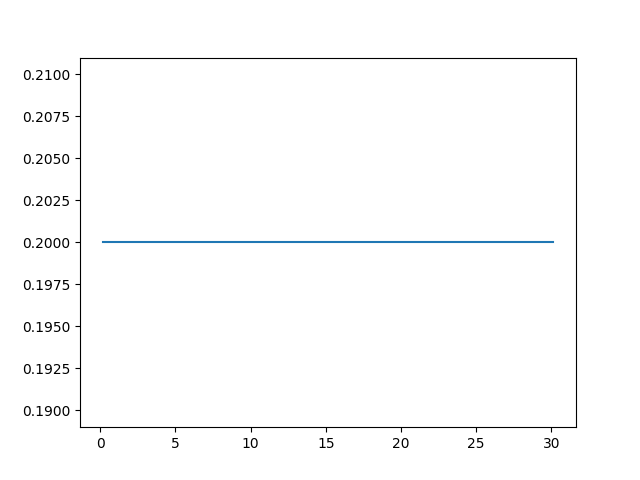
1. **Первый параметр**

**--lockbus N** запускает N рабочих процессов, которые быстро блокируют и увеличивают 64 байта случайно выбранной памяти из области mmap размером 16 МБ (только для процессоров Intel x86 и ARM). Это приведет к пропускам строк кэша и остановке работы процессоров.

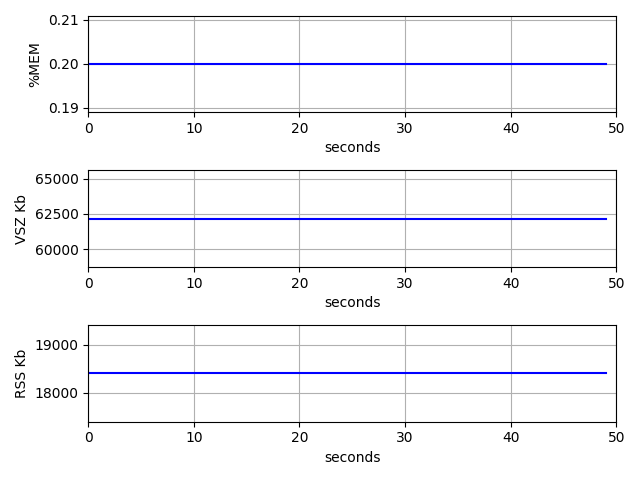
Команда запуска: stress-ng --lockbus 1

Запустим питон скрипт, который с помощью команды **top** считывает нагрузку на память и строит график.

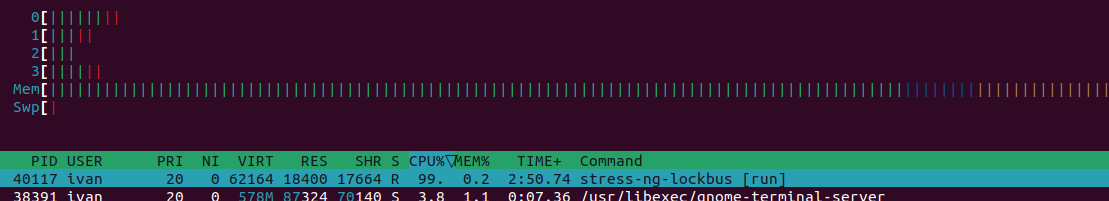
График:

Можем видеть, что потребление памяти константное. Это обуславливается тем, что запущенный стресс-тест блокирует фиксированный размер памяти.

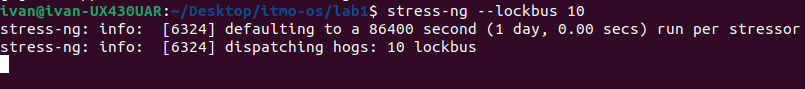
Посмотрим также на график утилиты pidstat:

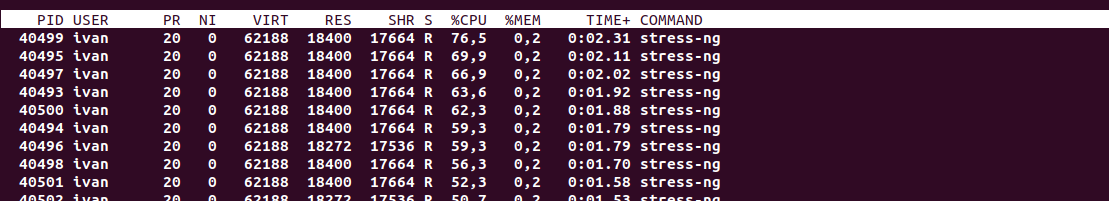


Посмотрим на данный процесс с помощью утилиты htop:

Имеем такие же значения — 0.2.

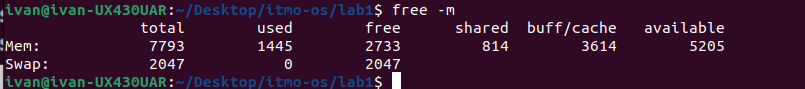
Попробуем запустить больше процессов:



Видно, что появились новые процессы, но каждый из них использует одинаковый процент памяти — 0.2

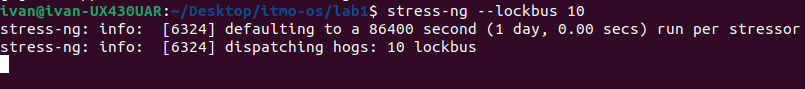
Проведем мониторинг утилитой free:

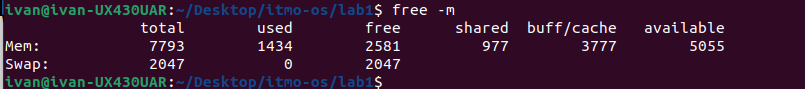
Посмотрим сколько свободно памяти, когда стресс тесты не работают:



free — 2733

А теперь запустим тесты и посмотрим изменения.



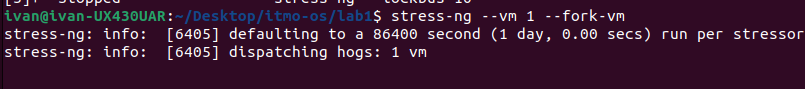
Ожидаемо значение параметра free уменьшилось.

1. **Второй параметр**

**--fork-vm** включает рекомендации по использованию виртуальной памяти, снижающие производительность, с помощью madvise на всех страницах разветвленного процесса.

Системный вызов madvise выдает предложения ядру об использовании постраничного ввода/вывода.

Команда запуска: stress-ng --vm 1--fork-vm



Запускаем один тест виртуальной памяти с функцией fork-vm

График (top):

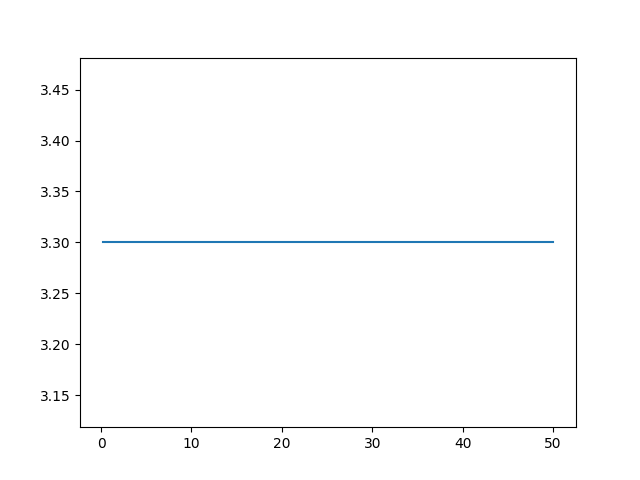
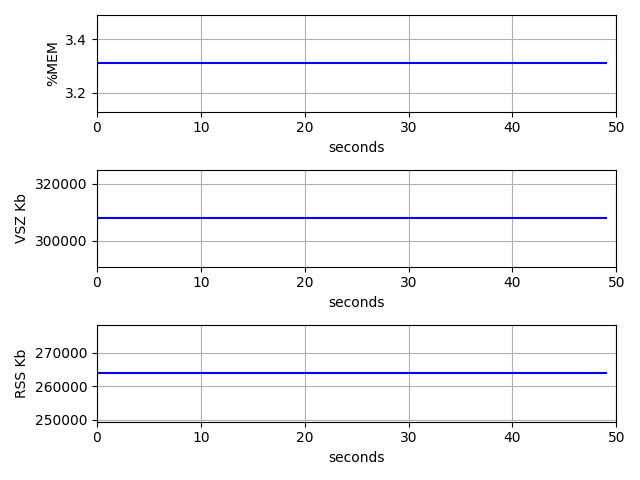


График (pidstat):



Запустим тесты еще раз и сравним выводы значения загрузки памяти утилит top и htop:



Значения одинаковы.

**3) Запуск с двумя параметрами**

Команда запуска: stress-ng --vm 1 **--**fork-vm --lockbus 1

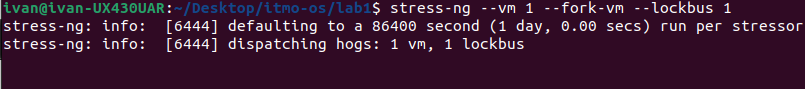
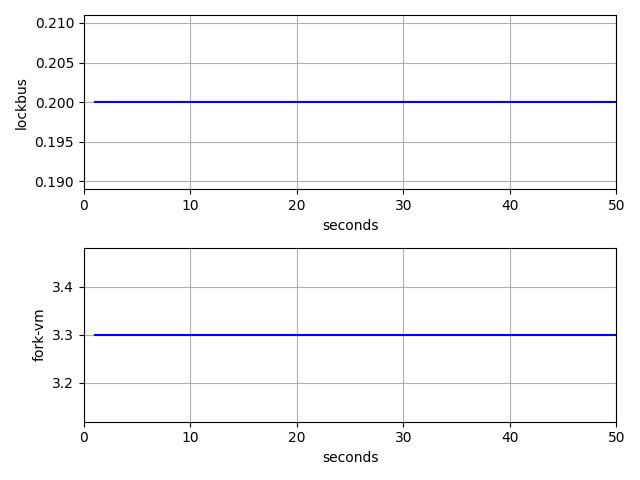


График:



**Выводы по мониторингу памяти:**

Был произведен мониторинг с помощью двух видов стресс-тестов. Первый параметр всегда выделял одинаковое количество памяти, второй же менял значения. При это процент загрузки памяти зависит от количества запущенных процессов работы с ней.

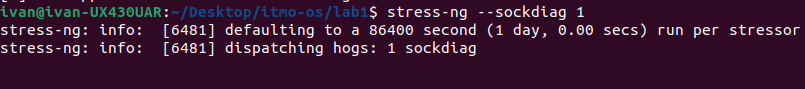
Утилиты top, pidstat и htop показали примерно равные результаты мониторинга.

## Network

1. **Первый параметр**

-- **sockdiag N** запускает N рабочих процессов, которые выполняют диагностику сетевых сокетов Linux sock\_diag (только для Linux). В настоящее время запрашивается диагностика с использованием UDIAG\_SHOW\_NAME, UDIAG\_SHOW\_VFS, UDIAG\_SHOW\_PEER, UDIAG\_SHOW\_ICONS, UDIAG\_SHOW\_RQLEN и UDIAG\_SHOW\_MEMINFO для семейства сокетных подключений AF\_UNIX.

Команда запуска: stress-ng --sockdiag 1

****

Запустим 3 питон скрипта, которые с помощью утилит bmon, ifstat и ip считают сумму полученных бит (RX) и переданных бит (TX) для сравнения утилит.

График bmon:

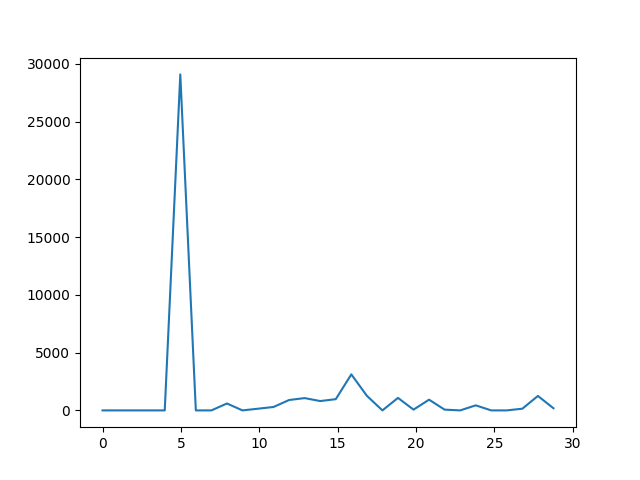


График ip:

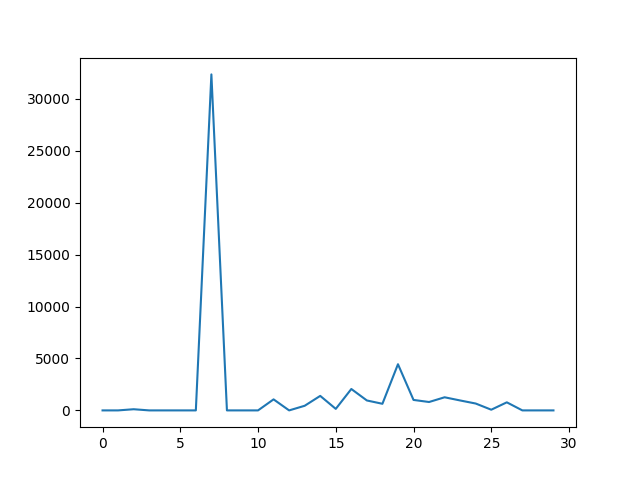
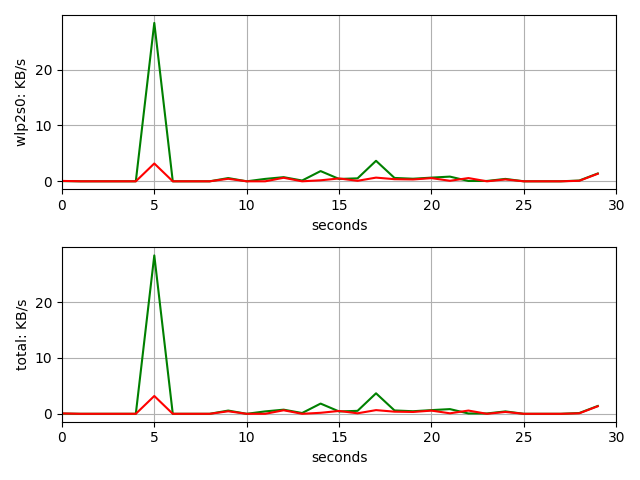


График ifstat:



Видим, что значения одинаковые. (Небольшое смещение по x обуславливается тем, что скрипты запускались вручную друг за другом — то есть была небольшая задержка)

Увеличим количество рабочих процессов:

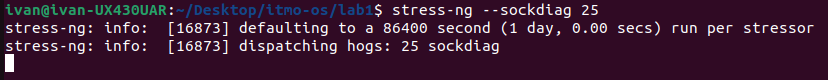
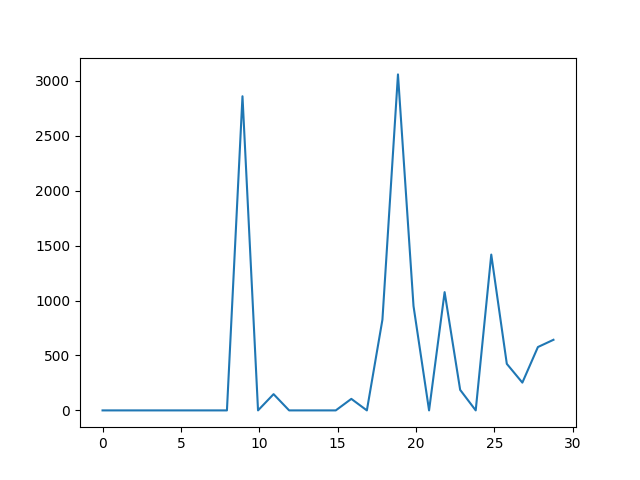


График:

****

Можно заметить, что с увеличением количества процессов, уменьшается нагрузка на сетевую подсистему.

Увеличение числа рабочих процессов может привести к уменьшению нагрузки на сетевую подсистему по нескольким причинам:

**1. Распределение нагрузки:** При увеличении числа рабочих процессов утилиты stress-ng, нагрузка на сетевую подсистему распределяется между этими процессами. Это означает, что каждый процесс будет обрабатывать меньше сетевых запросов и отправлять меньше данных, что в конечном итоге уменьшает общую нагрузку на сетевую подсистему.

**2. Конкуренция за ресурсы:** Увеличение числа рабочих процессов может привести к конкуренции за ресурсы, например, за доступ к сетевым портам или сетевому интерфейсу. Если есть только один процесс, он может монополизировать доступ к ресурсам и создавать высокую нагрузку на сетевую подсистему. Однако, с увеличением числа рабочих процессов, нагрузка становится более равномерной, что может привести к снижению нагрузки на сеть.

1. **Второй параметр**

**-- netlink-proc N** запускает N рабочих процессов, которые порождают дочерние процессы и отслеживают события процесса fork/exec/exit через коннектор proc netlink.

Команда запуска: sudo stress-ng --netlink-proc 1

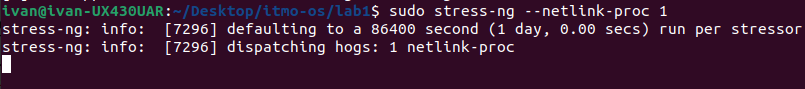


График:

График bmon:

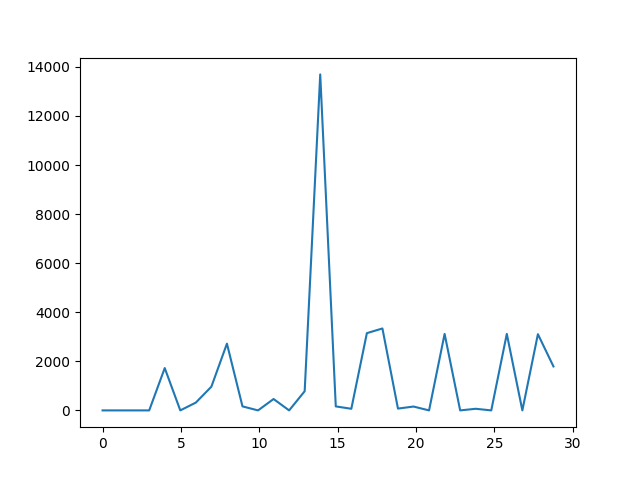


График ifstat:

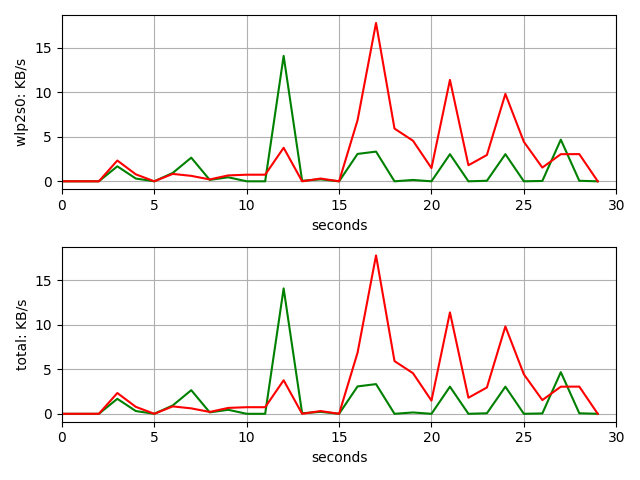
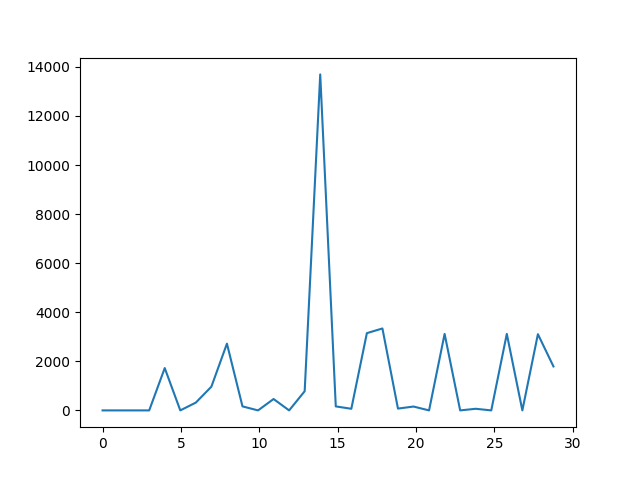


График ip:



Увеличим количество рабочих процессов:

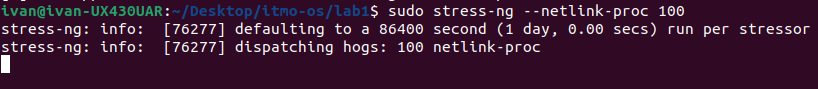
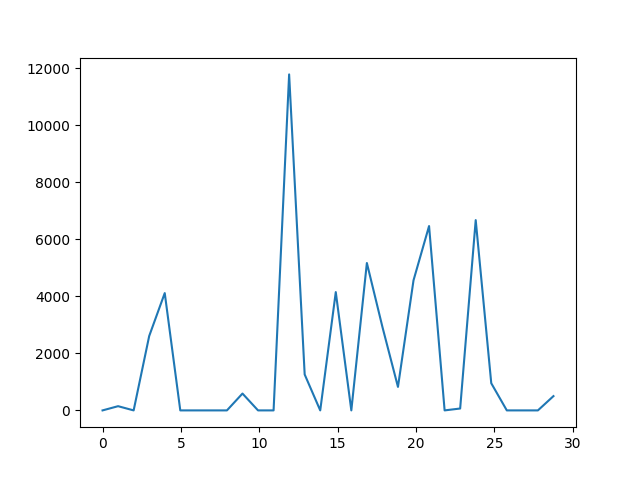


График:



Аналогично, увеличение процессов несет за собой уменьшение нагрузки сетевой подсистемы.

**3) Запуск с двумя параметрами**

Команда запуска: sudo stress-ng --netlink-proc 10 --sockdiag 10

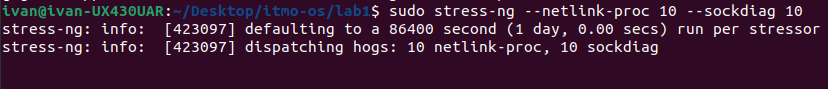
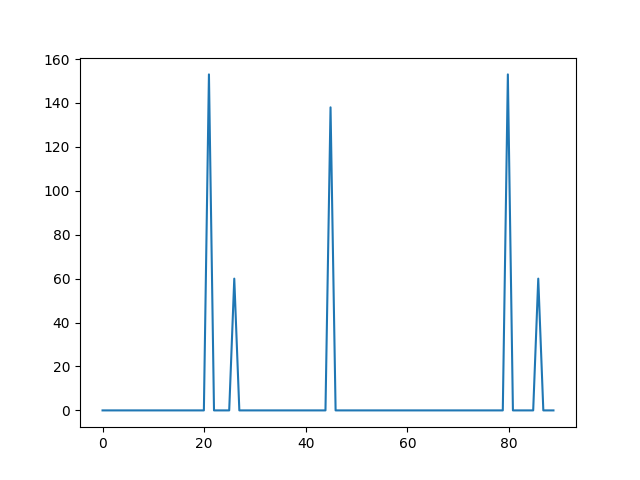


График:



Выводы по мониторингу сетевой подсистемы:

С увеличением количества процессов стресс-теста нагрузка на сетевую подсистему может уменьшаться по причинам, рассмотренным в 1 пункте.

Утилиты bmon, ifstat и ip показали практически одинаковые значения.

## Pipe

**1) Первый параметр**

**--pipe-size N** указывает размер канала в байтах

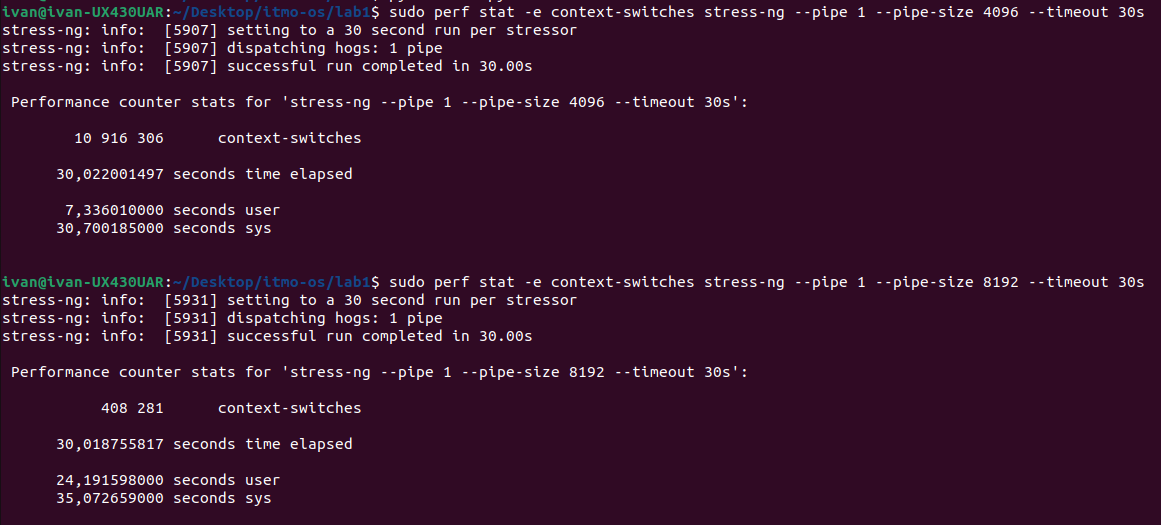
--**pipe N** запускает N рабочих процессов, выполняющих операции записи и чтения больших объемов данных по каналу. Осуществляется запись и чтение памяти, а также переключение контекста.

Команда запуска: stress-ng --pipe 1 --pipe-size 4096



Будем смотреть статистику по переключению контекста.

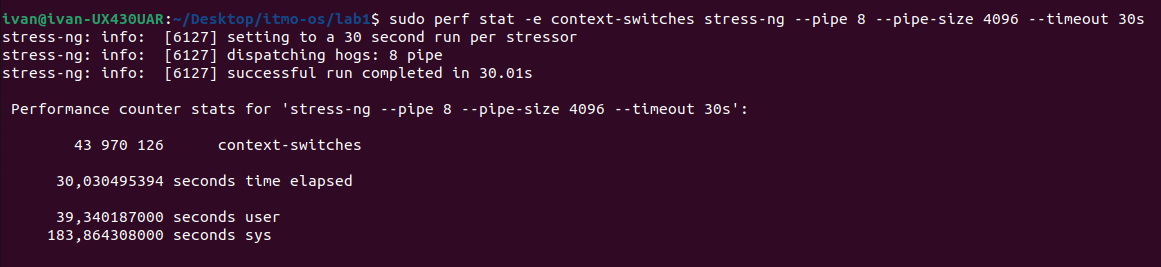
Для мониторинг context-switching воспользуемся утилитой perf:

sudo perf stat -e context-switches stress-ng --pipe 1 --pipe-size 4096 --timeout 30s

Можно заметить, что при увеличении размера канала количество переключений контекста уменьшается. При увеличении размера pipe (канала) в операционной системе, число context-switches может уменьшиться из-за оптимизации процесса ввода-вывода и управления процессами. Это может произойти:

1. Увеличение пропускной способности: Увеличение размера pipe может увеличить пропускную способность для передачи данных между процессами. Это означает, что больше данных может быть передано за один раз, что уменьшает необходимость в частых контекстных переключениях между процессами.
2. Снижение ожидания: Меньший размер pipe может привести к тому, что процессы чаще ожидают доступа к pipe, так как они должны чаще уступать CPU другим процессам. Увеличение размера pipe снижает вероятность ожидания и, следовательно, число контекстных переключений.
3. Увеличение эффективности: Увеличение размера pipe может сделать взаимодействие между процессами более эффективным, так как меньше времени будет затрачено на управление самим pipe и контекстными переключениями.

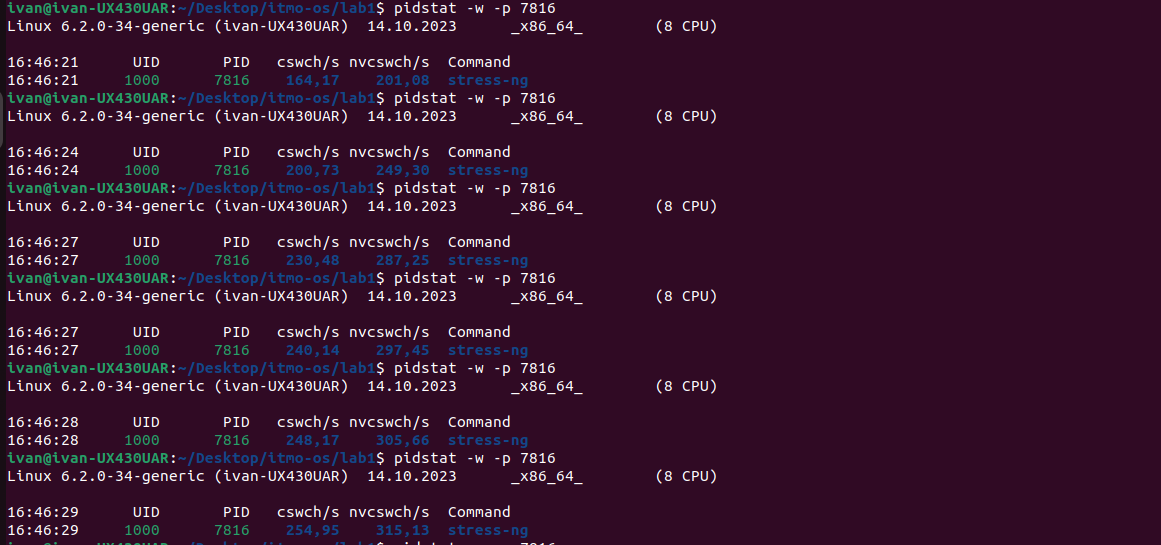
Однако, важно отметить, что слишком большой размер pipe также может иметь негативные эффекты, такие как использование большего объема памяти или потенциальные проблемы с производительностью.

Попробуем изменить количество тестов:

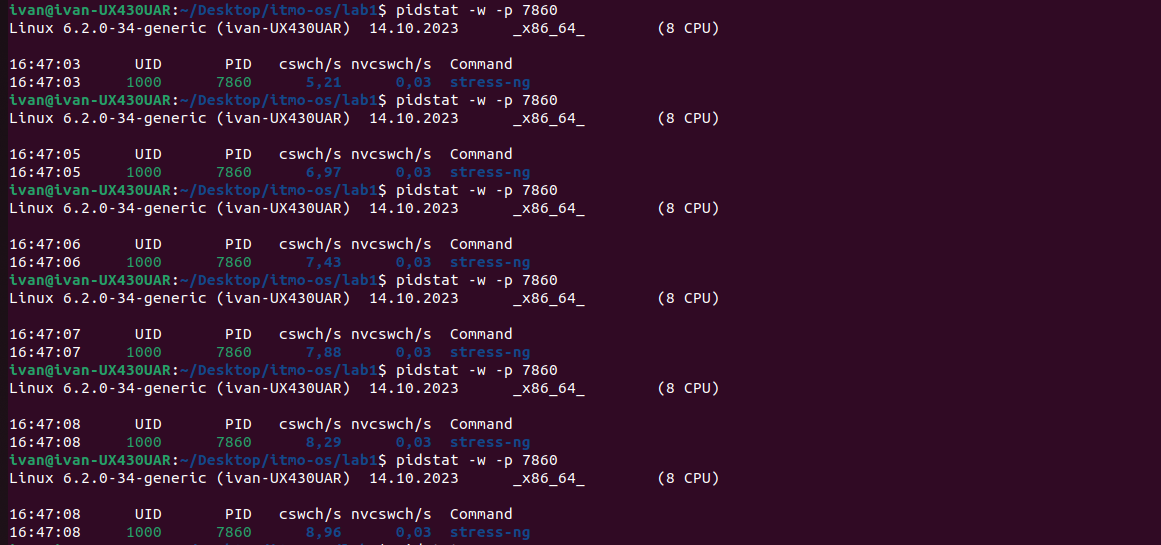
Количество стресс-тестов увеличивает количество переключений контекста.

Посмотрим на как параметры влияют на количество переключений контекстов в секунду с помощью утилиты pidstat:

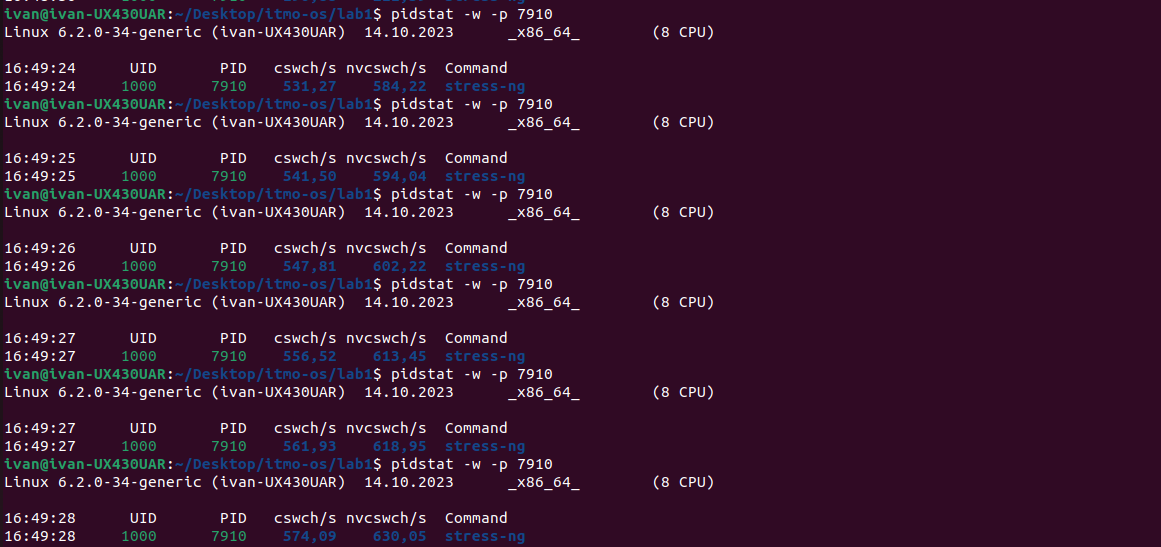
Запустим один тест:



Увеличим размер канала:



Запустим 8 тестов:

 Заметим, что увеличение размера канала также уменьшают количество переключений контекстов в секунду, но увеличение числа стресс-тестов увеличивает количество переключений контекстов в секунду.

**2) Второй параметр**

**-- pipeherd-yield** принудительно выполняет планирование после каждой записи, это увеличивает скорость переключения контекста.

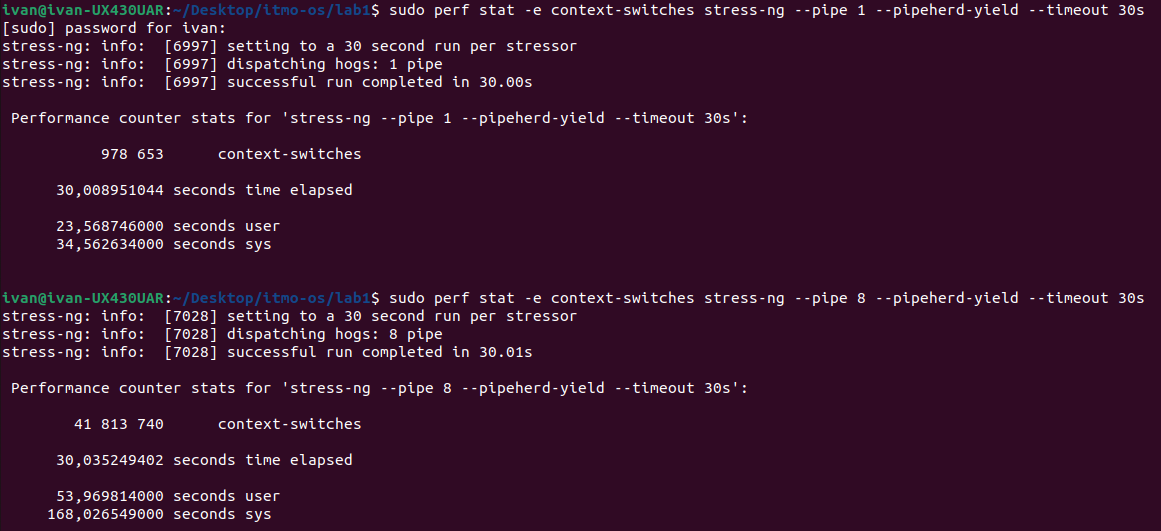
Команда запуска: stress-ng --pipe 1 --pipeherd-yield



Будем смотреть статистику по переключению контекста.

Для мониторинг context-switching воспользуемся утилитой perf:

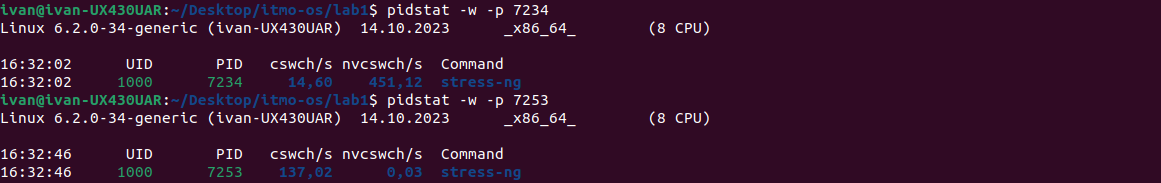
sudo perf stat -e context-switches stress-ng --pipe 1 --pipeherd-yield --timeout 30s



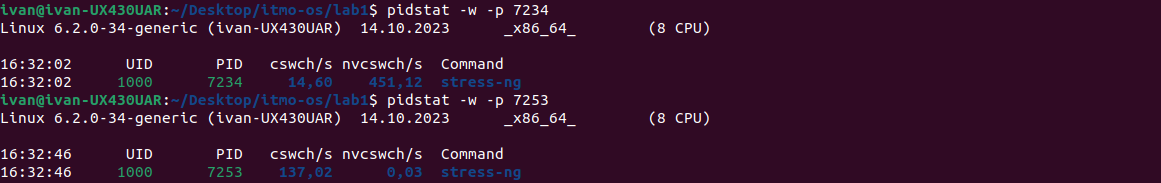
Количество стресс-тестов увеличивает количество переключений контекста.

Посмотрим на как параметры влияют на количество переключений контекстов в секунду с помощью утилиты pidstat:

Запустим один тест:

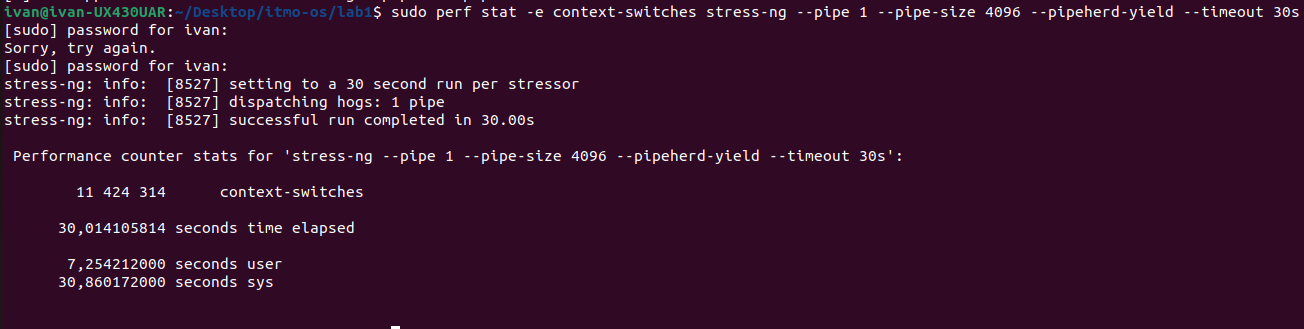


Запустим 8 тестов:

Можно увидеть, что количество переключений контекстов в секунду увеличивается при увеличении количества стресс-тестов.

**3) Запуск с двумя параметрами**

Команда запуска: sudo perf stat -e context-switches stress-ng --pipe 1 --pipe-size 4096 --pipeherd-yield --timeout 30s



**Выводы по мониторингу канальной подсистемы:** По результатам мониторинга можно сделать вывод, что увеличение размера канала уменьшают количество переключений контекстов в секунду, но увеличение числа стресс-тестов увеличивает количество переключений контекстов в секунду.

Утилиты pidstat и perf обе показали похожие данные и помогли прийти к одинаковым выводам.

## Sched

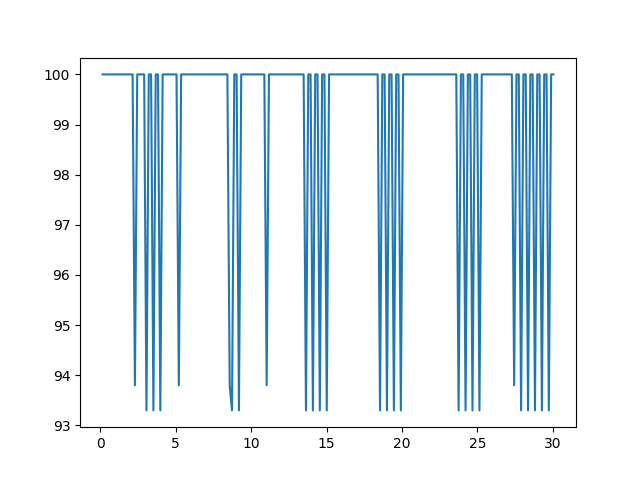
**1) Первый параметр**

-- sched-runtime - выбирает параметр времени выполнения для планировщика сроков . Значение по умолчанию — 99999 (в наносекундах).

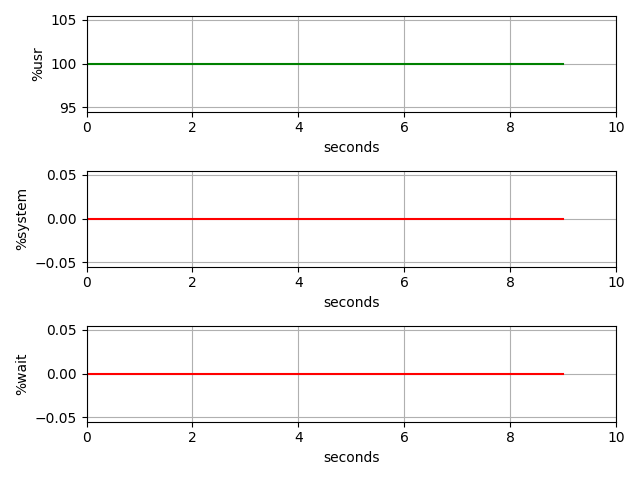
Команда запуска: stress-ng --cpu 1 --sched-runtime 5000



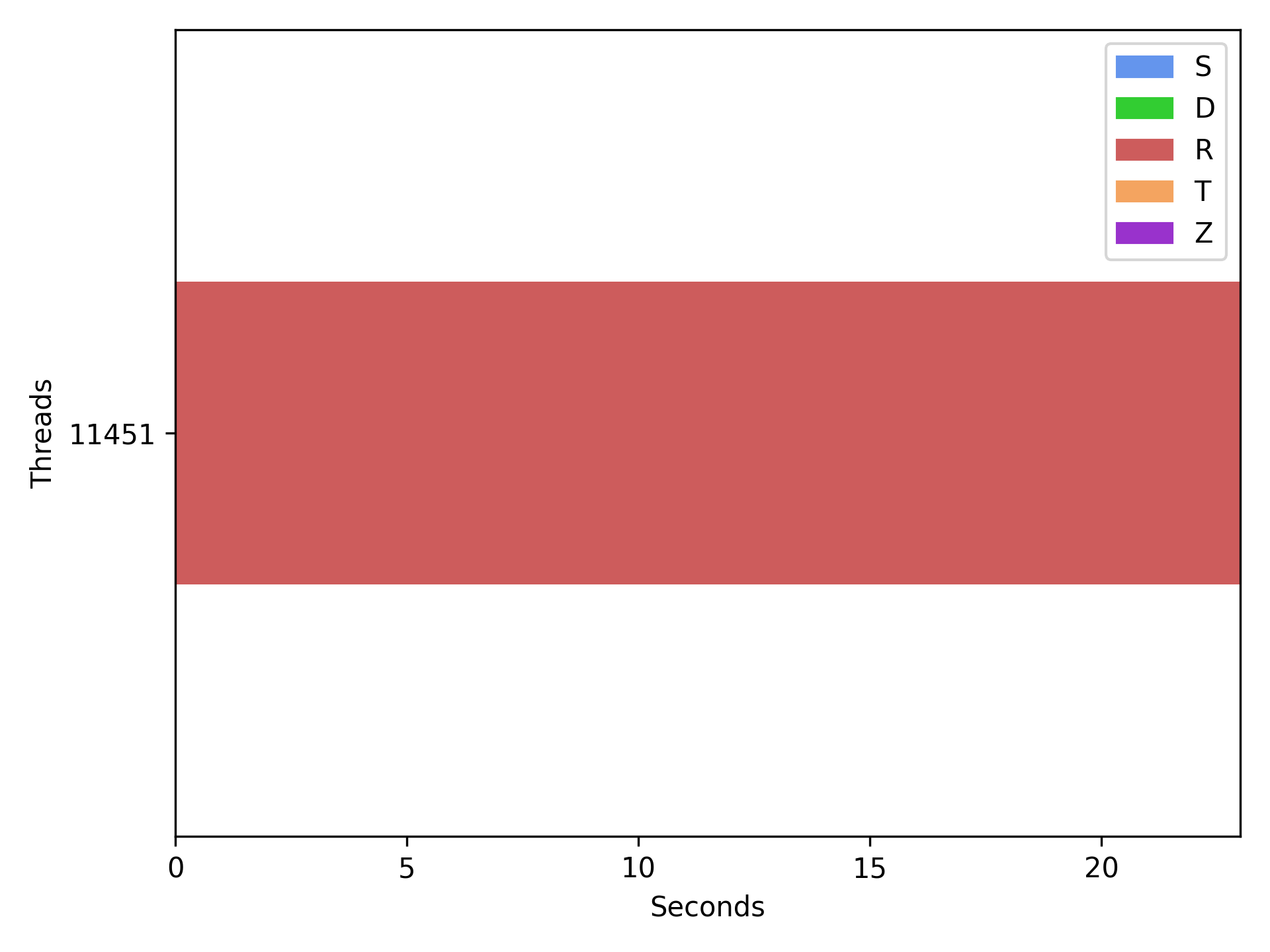
Построим график загрузки процессора утилитой top:



Посмотрим график процента ожидания утилитой pidstat:

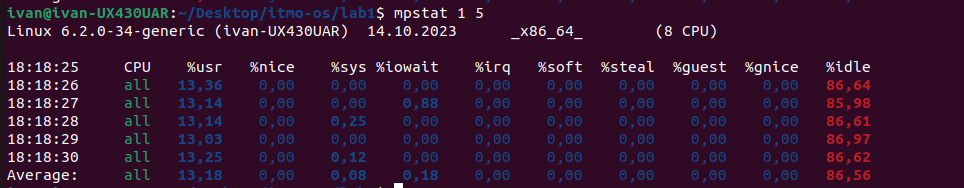
Процессор не простаивает в ожидании. Процесс загружает ядро на 100%, так же как и показала утилита top.

Посмотрим в каком состоянии находится процесс:



Всегда в Running.

Посмотрим также загрузку всего процессора утилитой mpstat:



Заметим, что процессор занят всего на 13% пользовательским процессом.

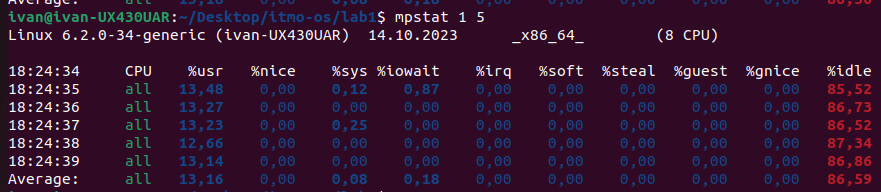
Также посмотрим статистику планировщика:

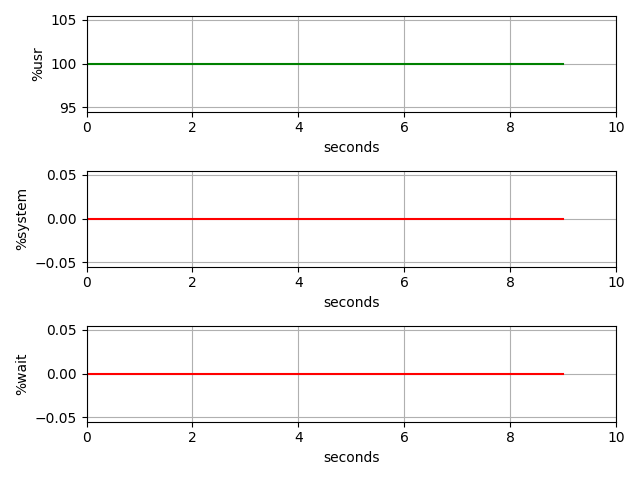


Первый параметр показывает время работы на процессоре, второй — время ожидания.

Попробуем увеличить время выполнения для планировщика:





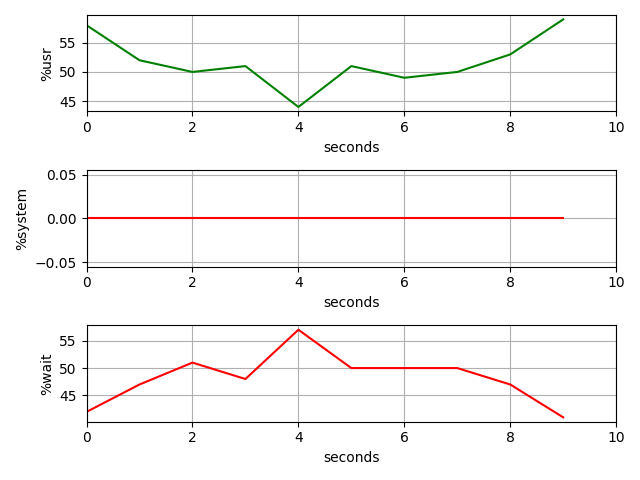
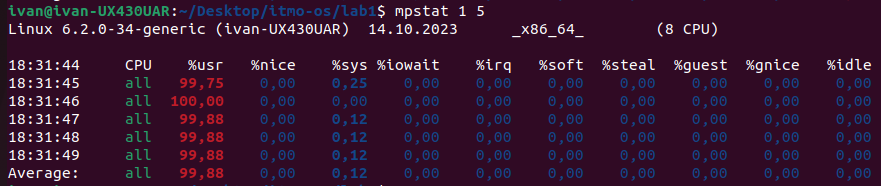




Статистика по нагрузке процессора не меняется, а вот время ожидания сокращается, так как увеличение этого параметра фактически увеличивает максимальное время выполнения процесса, что может привести к уменьшению времени ожидания процесса.

Попробуем запустить несколько процессов:





Ожидаемо увеличивается нагрузка на процессор и время ожидания процесса, также увеличивается и процент ожидания.

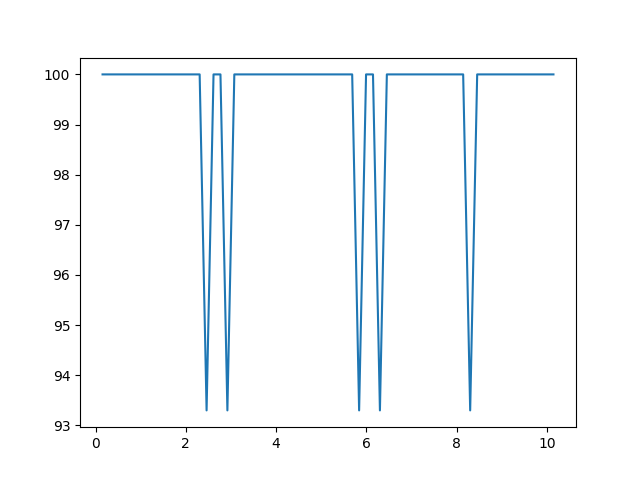
**2) Второй параметр**

--sched-prio выбирает уровень приоритета планировщика (только в Linux). Если планировщик не поддерживает это, то выбирается уровень приоритета по умолчанию 0.

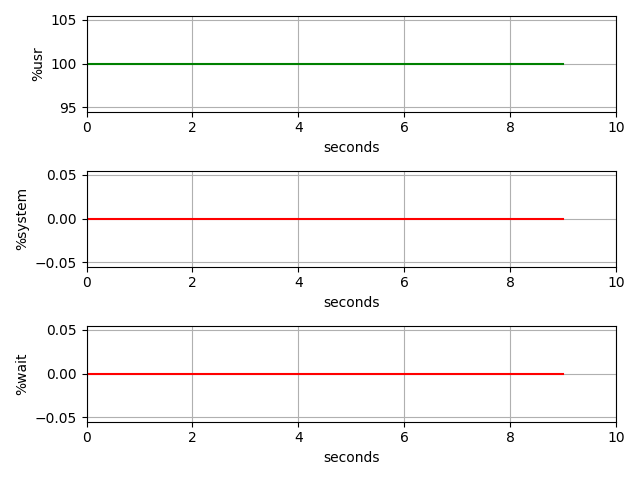
Команда запуска: stress-ng --cpu 1 --sched-prio 1



Построим график загрузки процессора утилитой top:

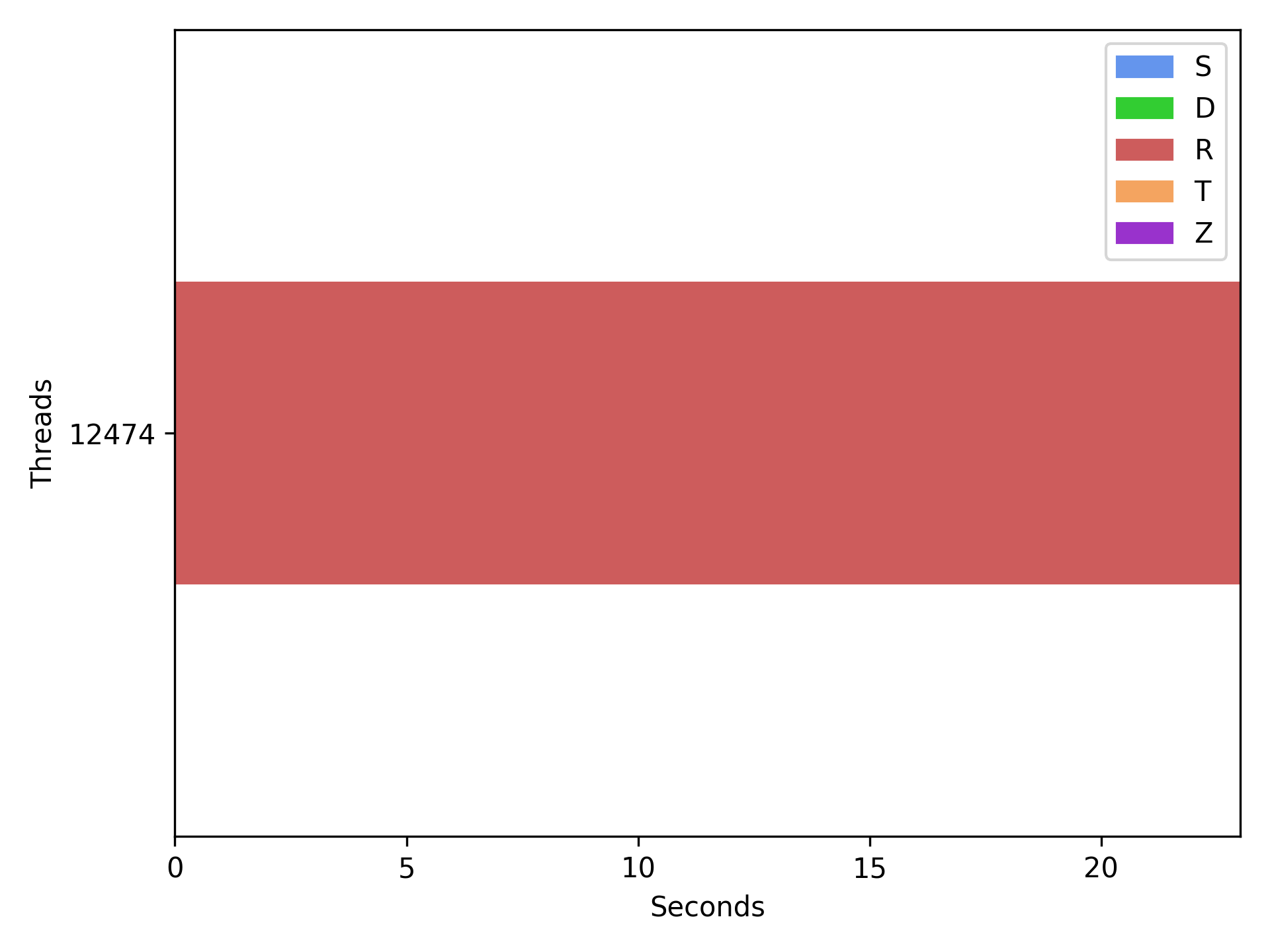


Посмотрим график процента ожидания утилитой pidstat:



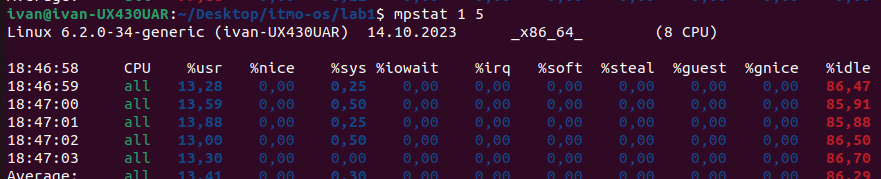
Процессор не простаивает в ожидании. Процесс загружает ядро на 100%, так же как и показала утилита top.

Посмотрим в каком состоянии находится процесс:



Всегда в Running.

Посмотрим также загрузку процессора утилитой mpstat:



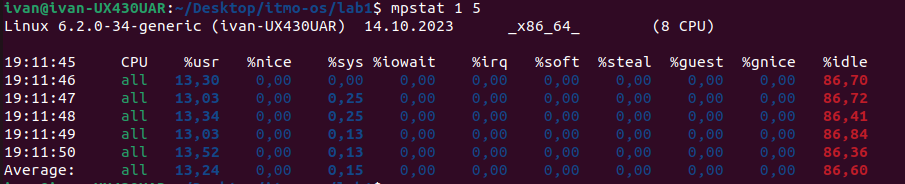
Аналогично заметим, что процессор занят всего на 13% пользовательским процессом.

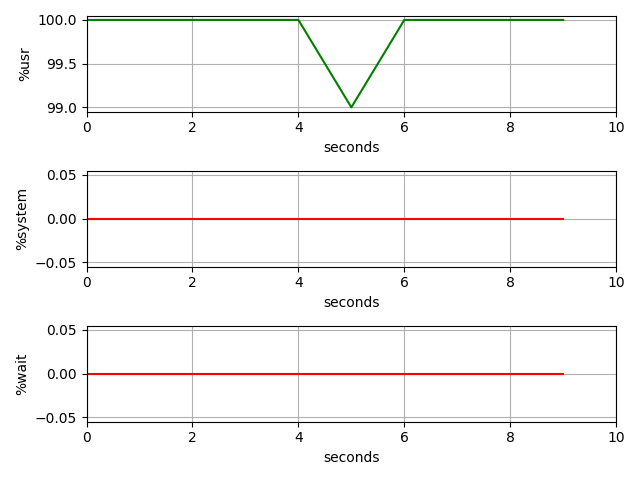
Также посмотрим статистику планировщика:



Попробуем увеличить приоритет:



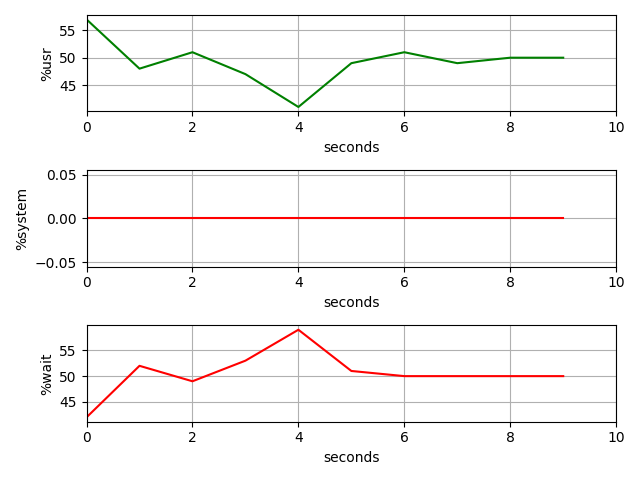


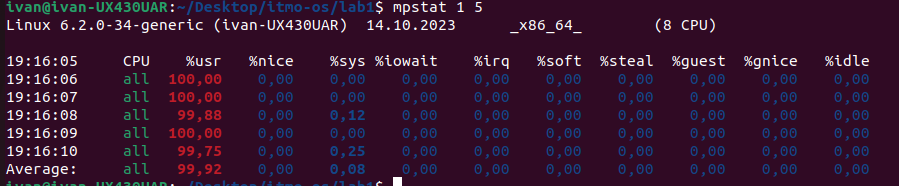


Аналогично нагрузка на процессор не меняется, но, так как повышен приоритет, уменьшается время ожидания.

Попробуем увеличить количество процессов:





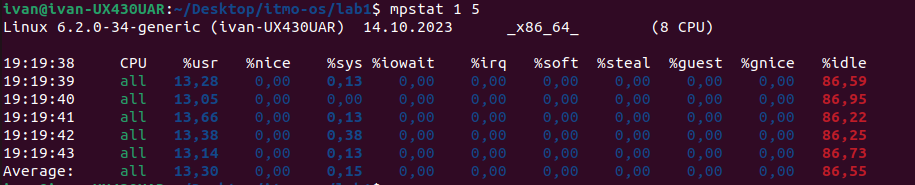


Ожидаемо, нагрузка увеличивается, время простоя процессора и время ожидания тоже, так как запускается много процессов.

**3) Запуск с двумя параметрами**

Команда запуска: stress-ng --cpu 1 --sched-prio 1 --sched-runtime 5000







**Выводы по мониторингу подсистемы планировщика:** С увеличением числа процессов, нагрузка на процессор увеличивается и время простоя процессора также увеличивается. На время ожидания процесса влияет его приоритет и время, выделенное на его выполнение.

# Выводы по лабораторной работе:

В ходе выполнения лабораторной работы я применил на практике утилиты для мониторинга процессов и системного анализа.