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

Верещагин Егор Сергеевич

Преподаватель:

Пашнин Александр
Денисович

Санкт-Петербург, 2023

Задание

Лабораторная работа №1

Внимание! В связи с поздним опубликованием лабораторной работы, вторая часть лабораторной работы (реальный бенчмарк) удалена из задания.

Основная цель лабораторной работы — знакомство с системными инструментами анализа производительности и поведения программ. В данной лабораторной работе Вам будет предложено произвести нагрузочное тестирование Вашей операционной системы при помощи инструмента stress-ng.

В качестве тестируемых подсистем использовать: cpu, cache, io, memory, network, pipe, scheduler.

Для работы со счетчиками ядра использовать все утилиты, которые были рассмотренны на лекции (раздел 1.9, кроме kdb)

Ниже приведены списки параметров для различных подсистем (Вам будет выдано 2 значения для каждой подсистемы согласно варианту в журнале). Подбирая числовые значения для выданных параметров, и используя средства мониторинга, добиться **максимальной** производительности системы (BOGOPS, FLOPS, Read/Write Speed, Network Speed).

Код

https://github.com/zula-baldez/os

Вариант:
cpu: [union,idct];

cache: [prefetch-l3-size,cache-ways];

io: [ioprio,ioport];

memory: [lockbus,mmaphuge-mmaps];

network: [netlink-task,sockdiag];

pipe: [sigpipe,pipeherd-yield]; sched: [resched,sched-runtime]

CPU:

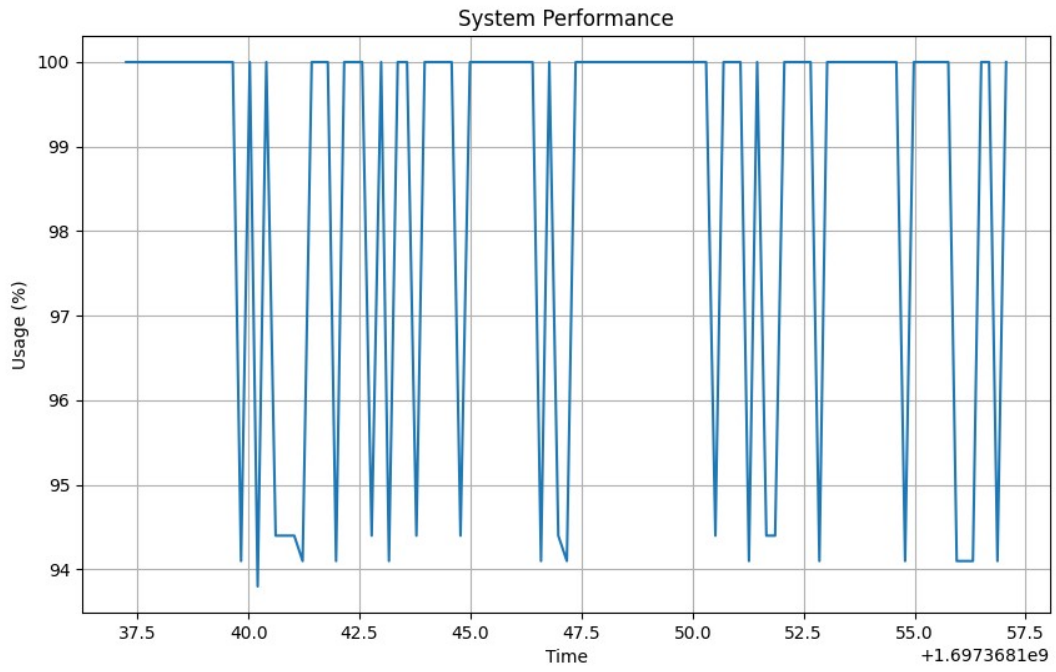
1) union.

Выполняет целочисленную арифметику над смесью битовых полей в union C. Это определяет, насколько хорошо компилятор и ЦП могут выполнять загрузку и сохранение целочисленных битовых полей.

Несколько раз запустим команду `stress-ng --cpu {i} --cpu-method union --timeout 20 —metrics-brief`

и посмотрим на графики нагруженности центрального процессора, информацию о нагруженности будет получать утилитой top с параметрами -b -n — батч мод и укажем, сколько раз выполнить команду.

Для одного процесса



Также используем pidstat

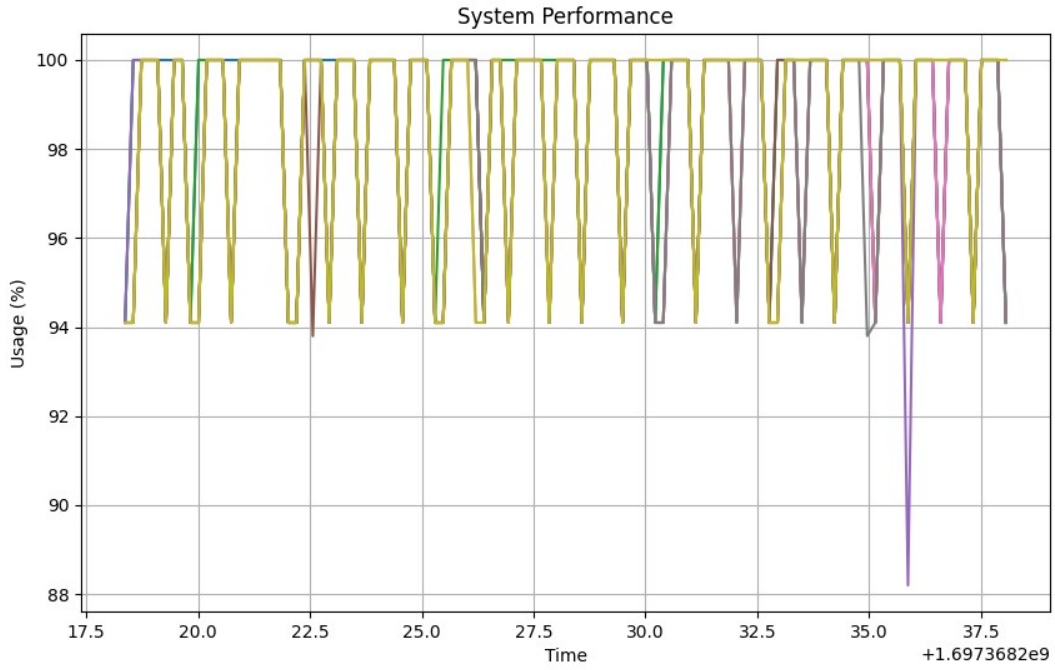
Также посмотрим на потребление CPU с помощью команды pidstat:

```
22:50:00      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
22:50:01     1000    671726  100,00    0,00    0,00    0,00  100,00     5  stress-ng
22:50:02     1000    671726  100,00    0,00    0,00    0,00  100,00     5  stress-ng
22:50:03     1000    671726  100,00    0,00    0,00    0,00  100,00     5  stress-ng
22:50:04     1000    671726  100,00    0,00    0,00    0,00  100,00     4  stress-ng
22:50:05     1000    671726  100,00    0,00    0,00    0,00  100,00     8  stress-ng
```

Данные такие же

Теперь посмотрим на программу при другом количестве процессов(например, 9)

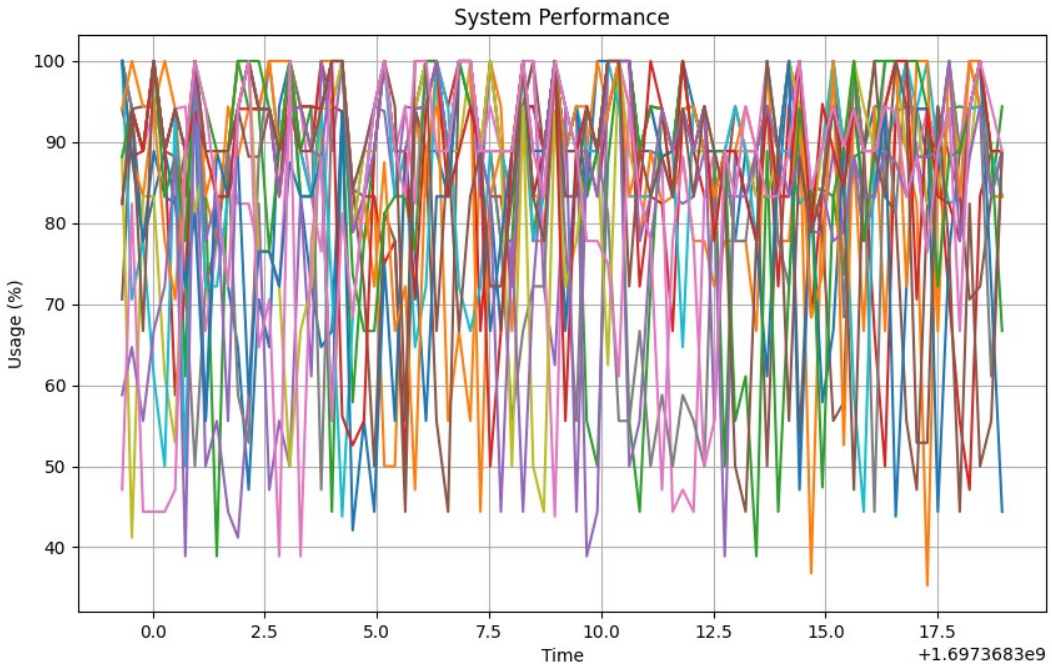
top:



Видим, что процессы используют почти по 100 процентов ядра, на котором они работают, результаты pidstat такие же

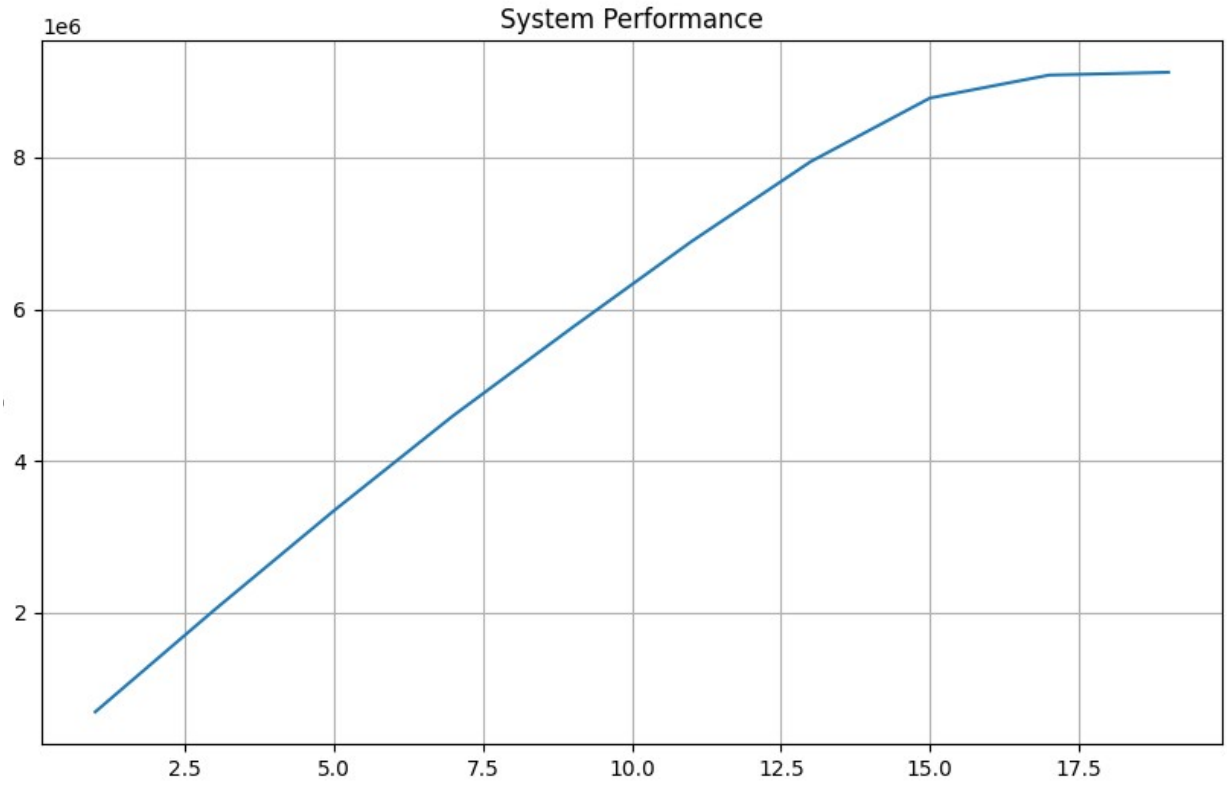
```
egor@OMEN-Laptop-15-en0xxx:~$ pidstat -p 671952 1
Linux 6.2.0-34-generic (OMEN-Laptop-15-en0xxx)  17.10.2023      _x86_64_        (16 CP

22:54:13      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
22:54:14     1000    671952  100,00    0,00    0,00    0,00  100,00     1  stress-ng
22:54:15     1000    671952  100,00    0,00    0,00    0,00  100,00     1  stress-ng
```

Попробуем теперь 17 процессов



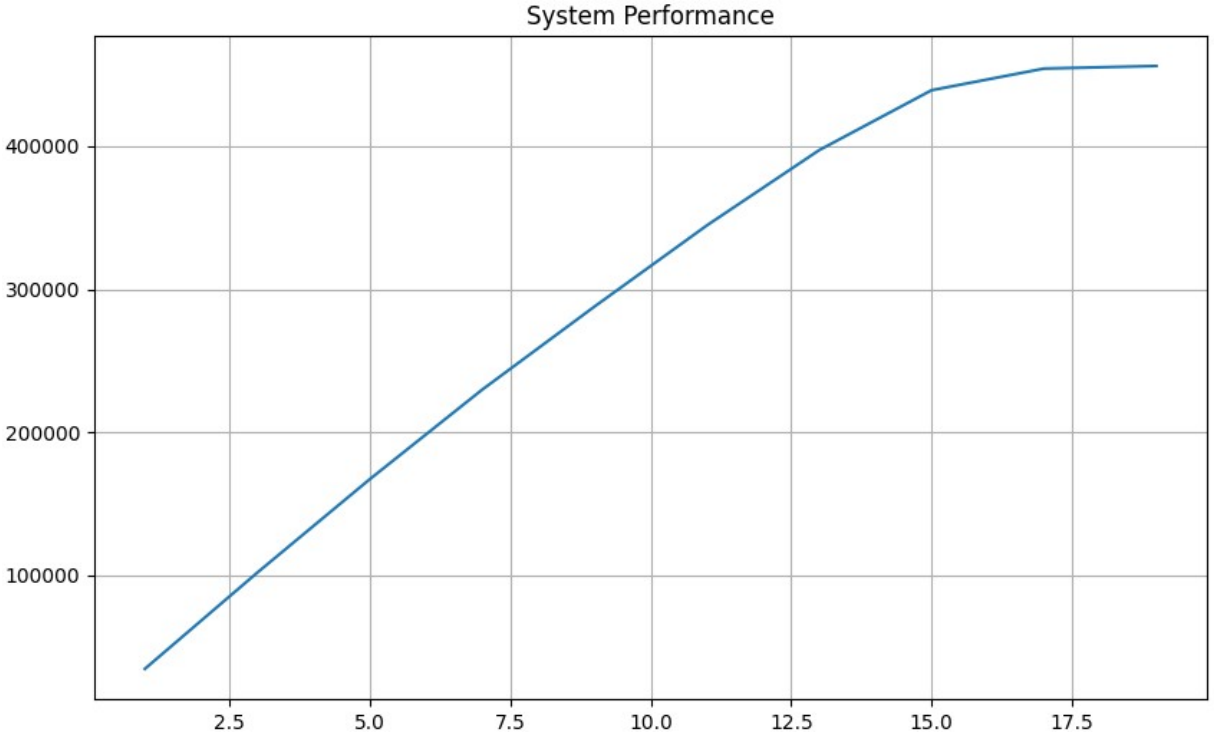
Этот результат объясняется тем, что в системе 8 ядер и 2 гипертреда. 17 процессов банально не могут выполняться одновременно, поэтому шедулеру приходится периодически менять выполняемые процессы.

Посмотрим также на график bogoops:



Интересно, что после 16 процессов количество операций не увеличивается — процессор достигает максимальной производительности.

bogoopsp/s похож

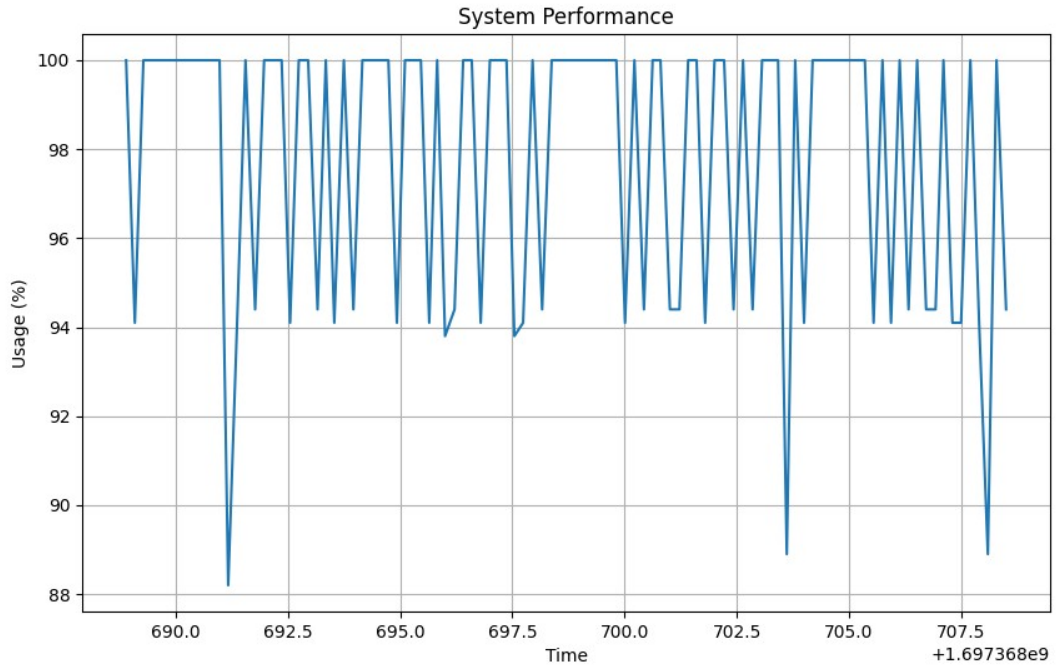


2) idct

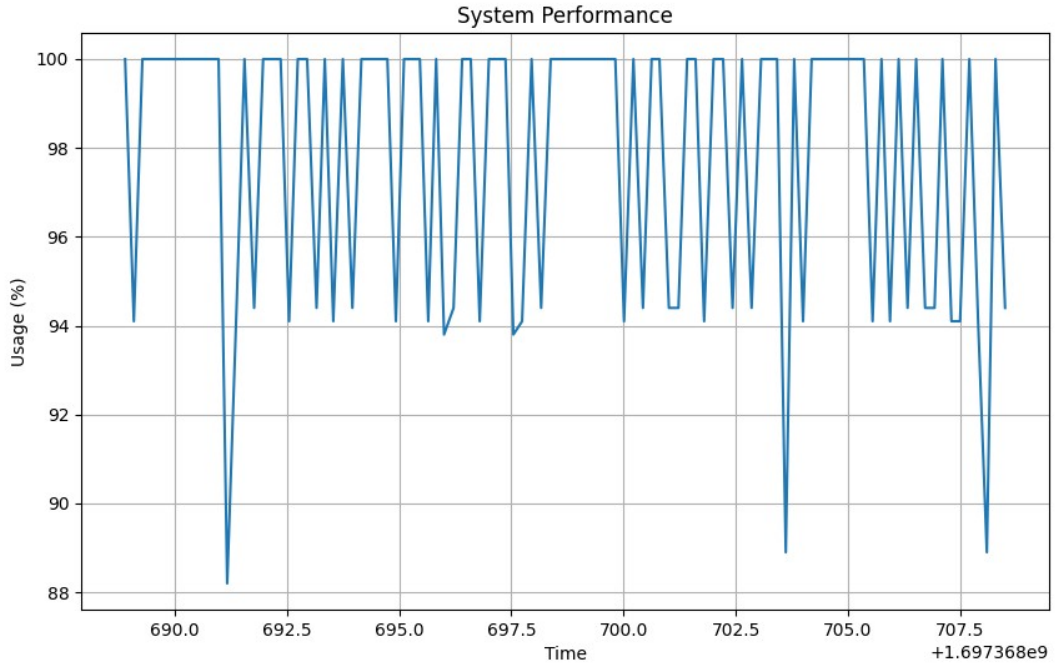
man - 8 × 8 IDCT (обратное дискретное косинусное преобразование).

Аналогично запустим разное количество процессов и посмотрим графики bogops

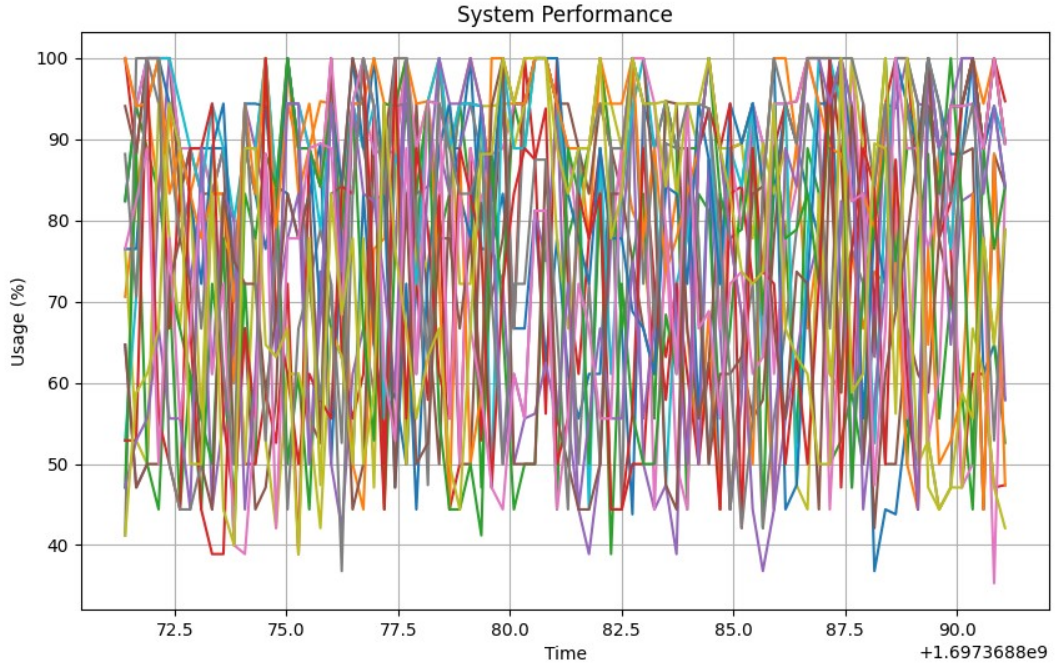
1 процесс



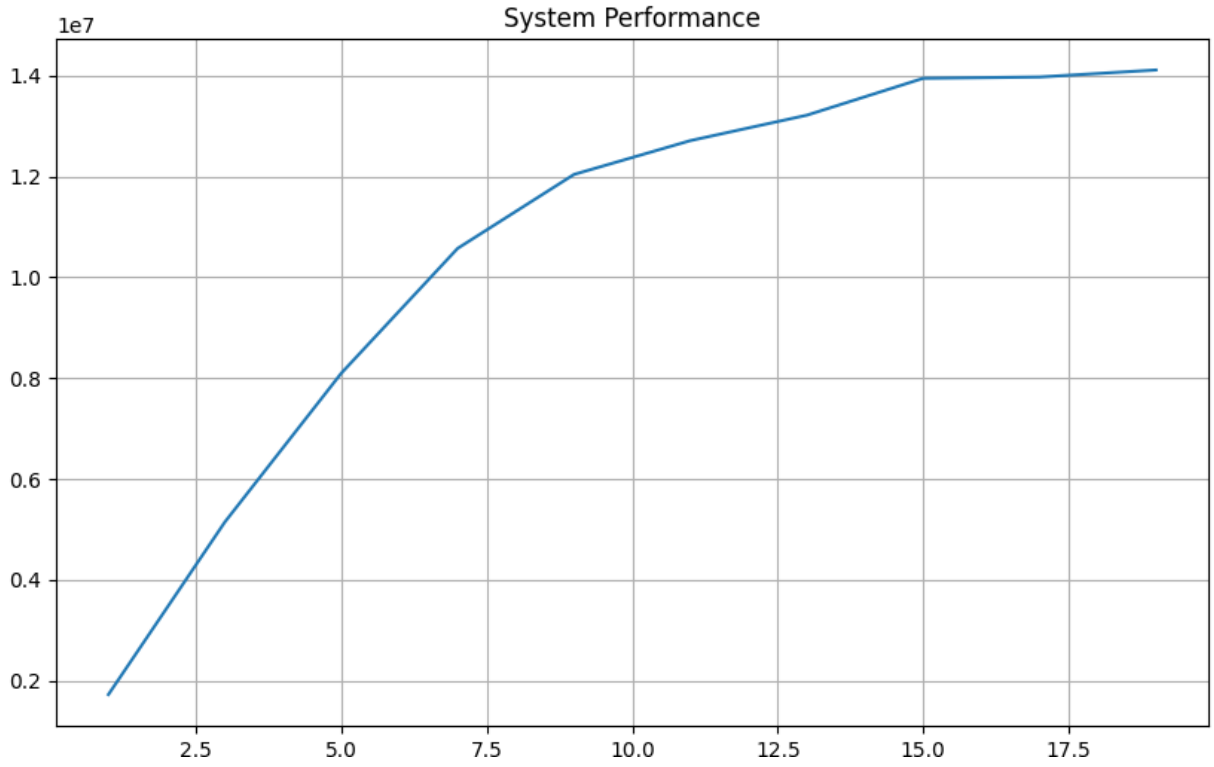
9 процессов



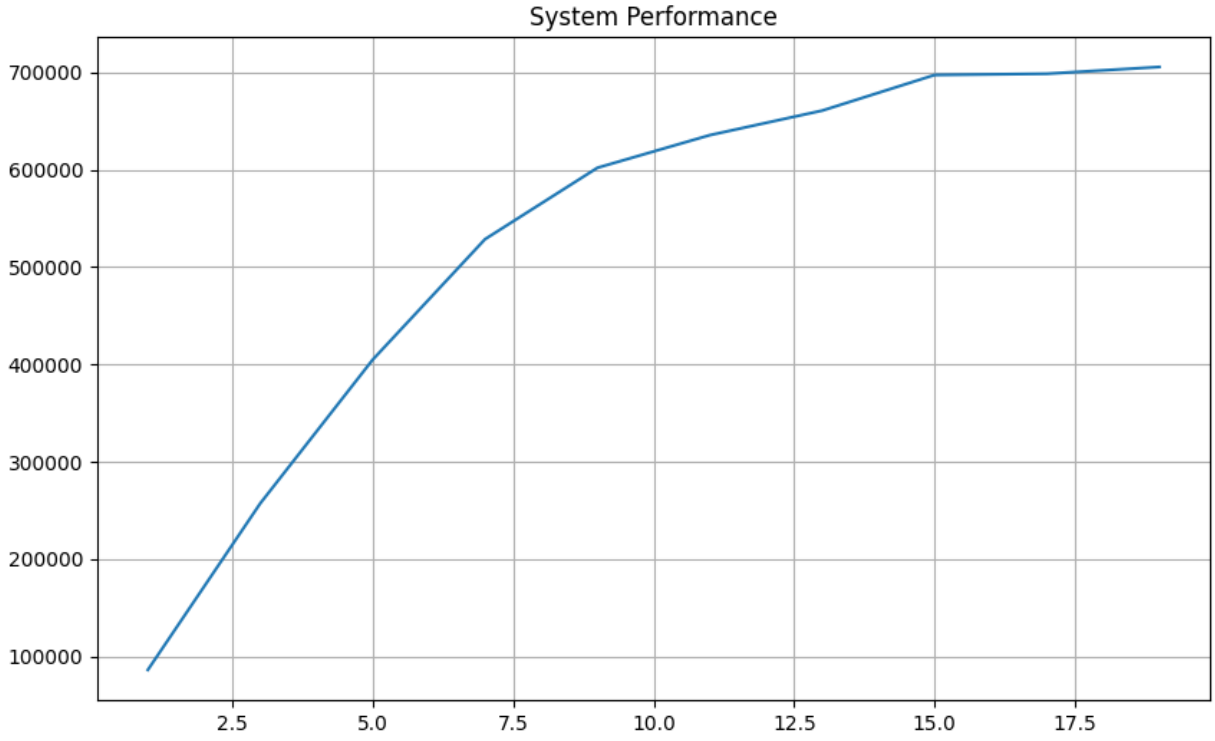
19 процессов



bogops


System Performance
1e7
1.4
1.2
1.0
0.8
0.6
0.4
0.2
2.5
5.0
7.5
10.0
12.5
15.0
17.5

bogops p/s


System Performance
700000
600000
500000
400000
300000
200000
100000
2.5
5.0
7.5
10.0
12.5
15.0
17.5

Результаты идентичные

Cache:

1) **-cache-ways**

--cache-ways N — указывает, сколько строк кеша содержит каждый cache set.

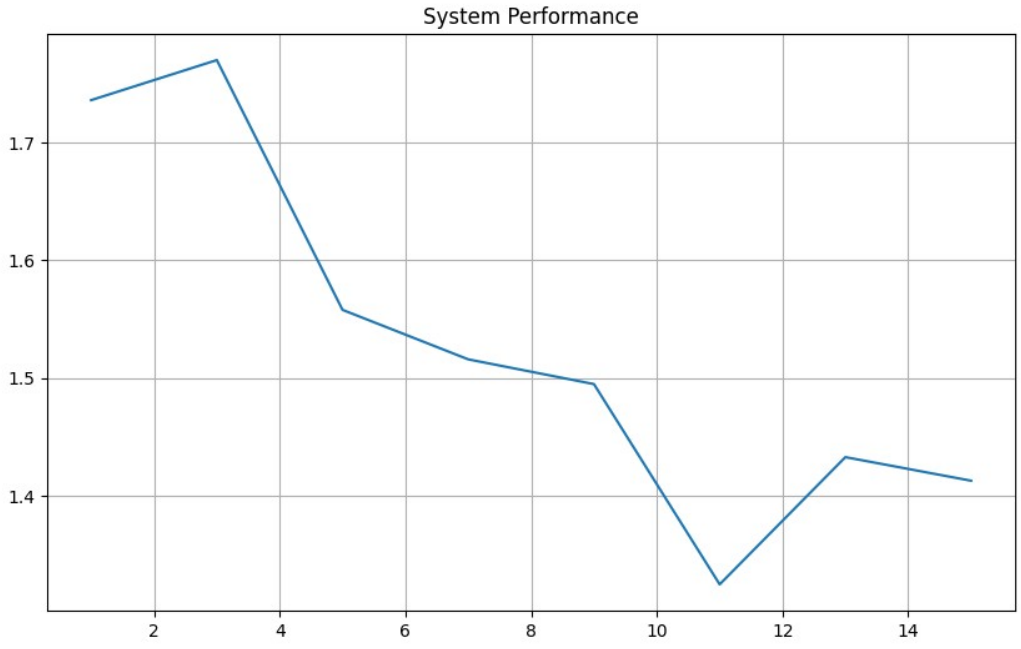
Для мониторинга кеша будем использовать perf для получения информации об обращениях к кешу и кеш промахах.

Команда:

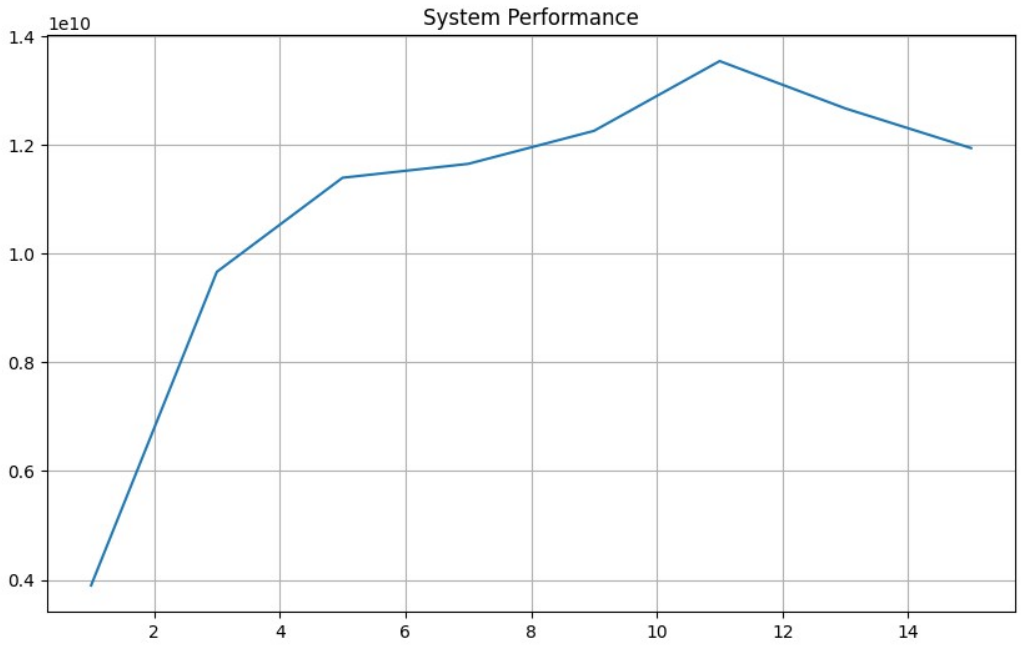
```
sudo perf stat -e cache-references,cache-misses stress-ng --cache 1 --cache-ways {i} --timeout 20
```

Запустим программу с разными значениями и посмотрим тенденции

Промахи(процент)



Обращения к кешу



Количество обращений к кешу растет, а вот процент промахов уменьшается. Дело в том, что кешируется не только непосредственно адресуемая память, но и память после нее(зависит от параметра N cache-ways).

Также посмотрим на статистку использования кеша до и после

```
egor@OMEN-Laptop-15-en0xxx:~$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 2289188 841616 5425144    0    0   628  2735   46   93  5  5 86  4
 0
egor@OMEN-Laptop-15-en0xxx:~$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 2224324 842160 5438200    0    0   615  2677   54  110  5  5 87  4
 0
```

Память, занимаемая кешем, увеличилась

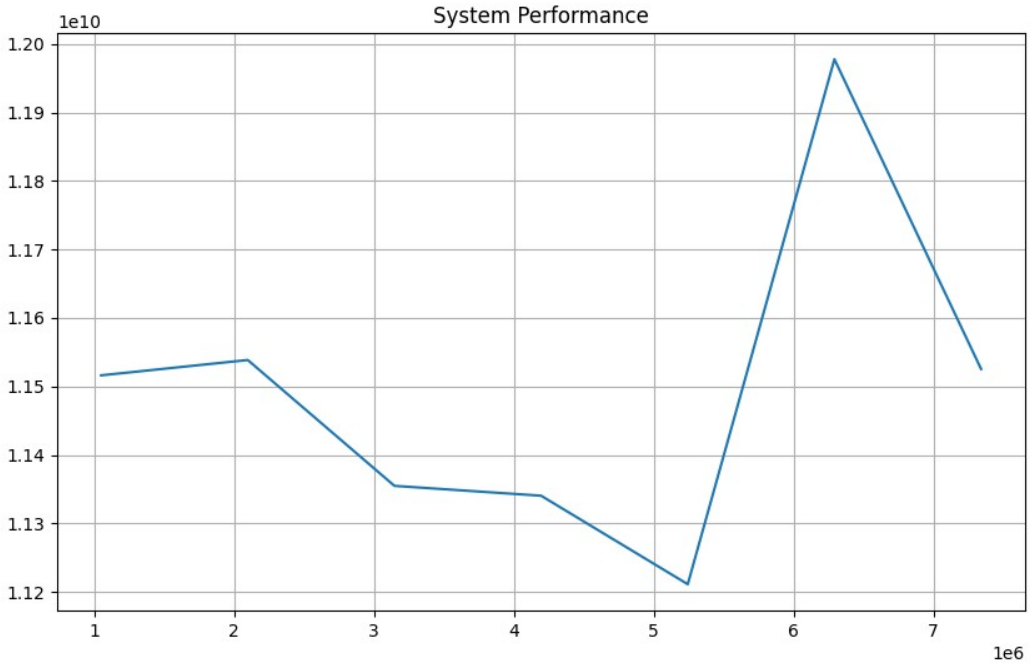
2) prefetch-l3-size

указывает размер l3 кеша

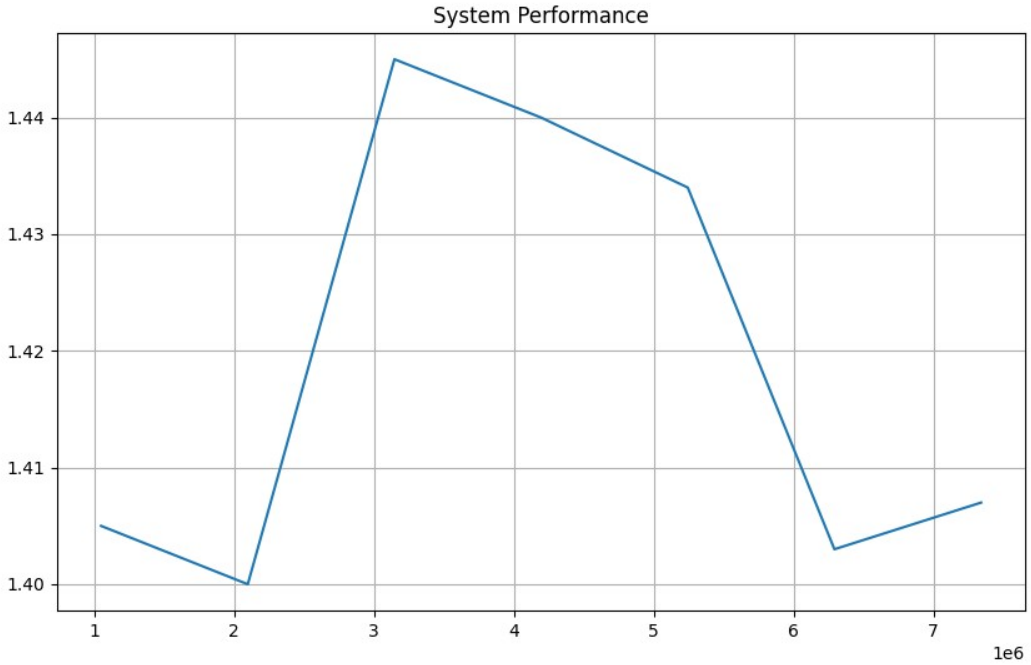
Мониторить будем абсолютно также, утилитой perf будем мониторить промахи и обращения

Статистика утилитой perf:

Обращения



Промахи



L3 — общий кеш. При быстром запуске одного процесса размер l3 особенно не влияет на промахи.

IO:

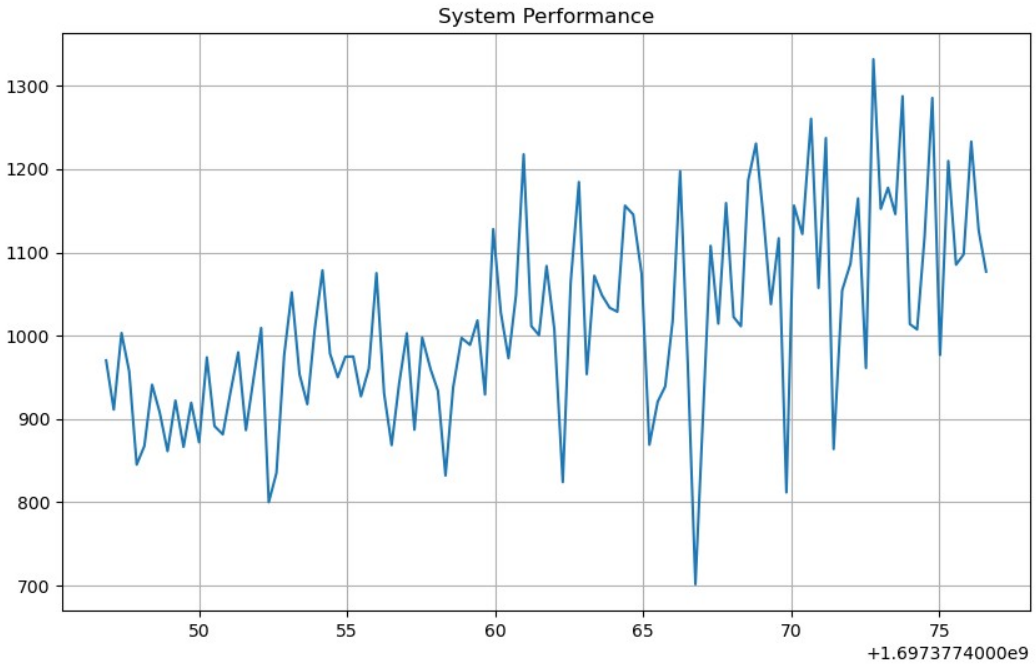
1) Ioprio

Запускает N потоков, которые выполняют системные вызовы ioprio_get, ioprio_set

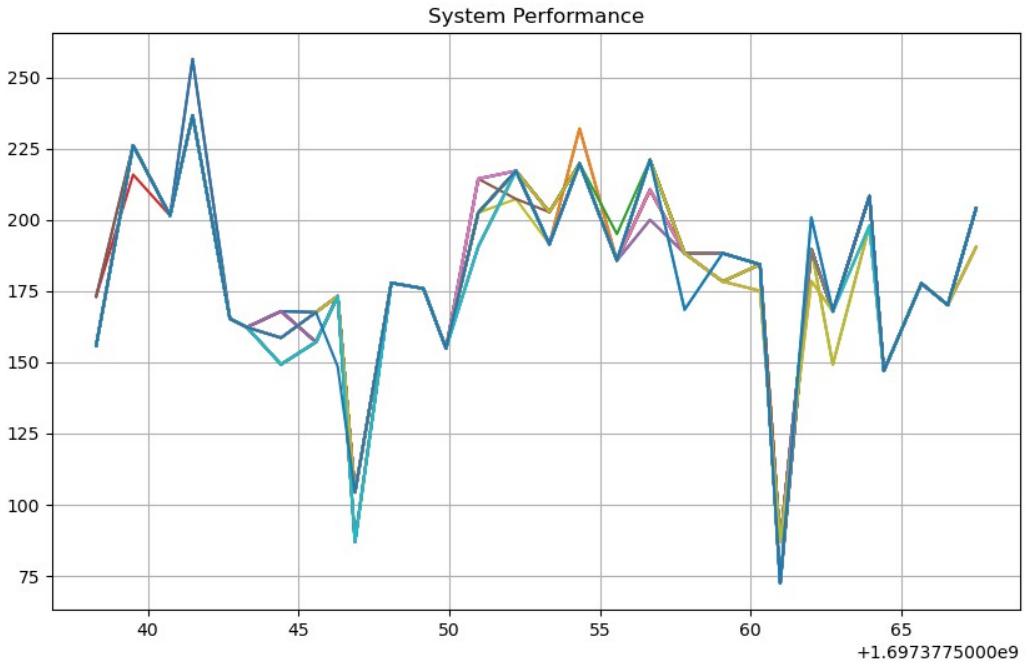
Будем мониторить команду `stress-ng --ioprio {i} --timeout 30`

с помощью iotop -b -P -n 1 промониторим скорость записи

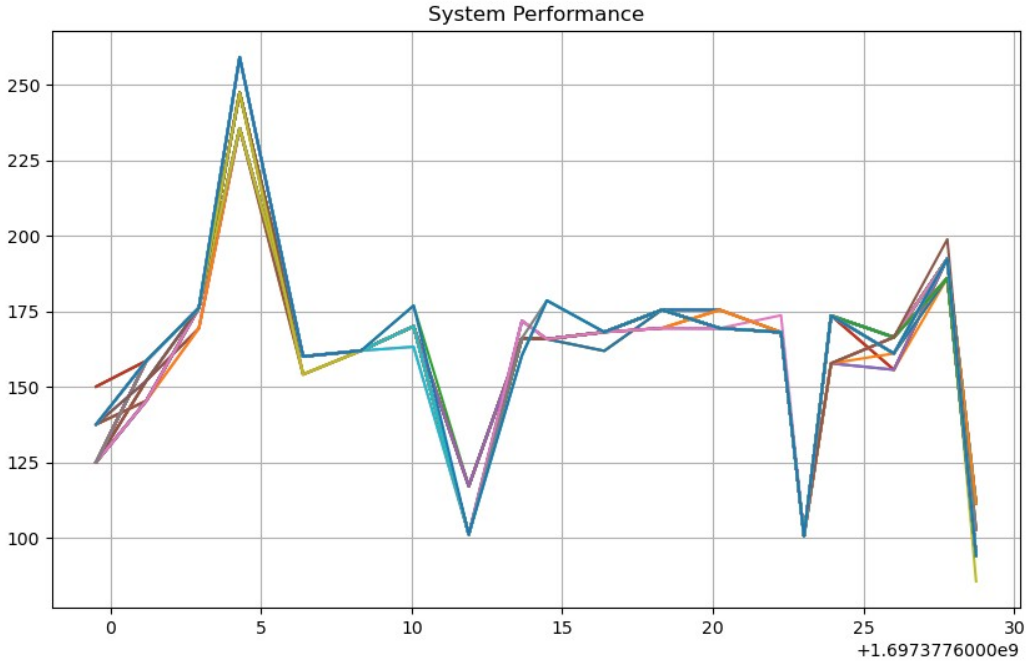
1 процесс



31 процесс


System Performance
250
225
200
175
150
125
100
75
40
45
50
55
60
65
+1.6973775000e9

51 процесс

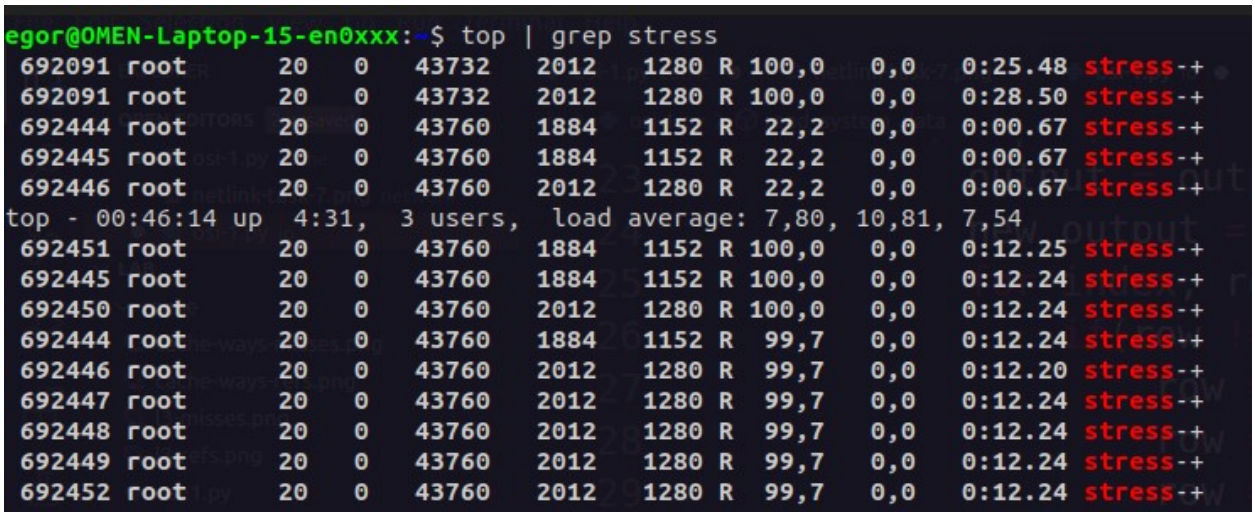

System Performance
250
225
200
175
150
125
100
0
5
10
15
20
25
30
+1.6973776000e9

Как видно, средняя скорость чтения снижается. Это объясняется тем, что процессы упираются в верхний лимит скорости чтения

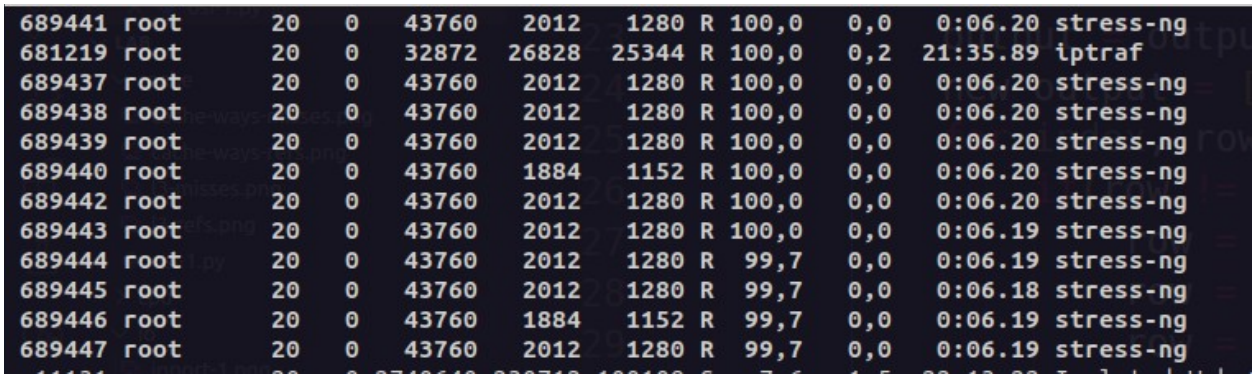
2) ioport

запускает N рабочих процессов, а затем выполняет пакеты из 16 операций чтения и 16 операций записи ioport 0x80 (только для систем Linux x86). Ввод-вывод, выполняемый на платформах x86 через порт 0x80, приведет к задержкам в ЦП, выполняющем ввод-вывод.

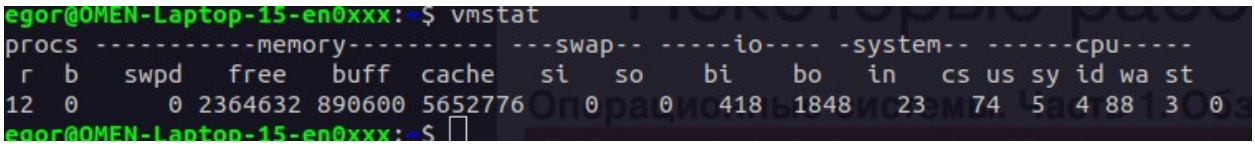
Скорость нулевая



Однако процессы грузятся на 100%



Если заглянуть



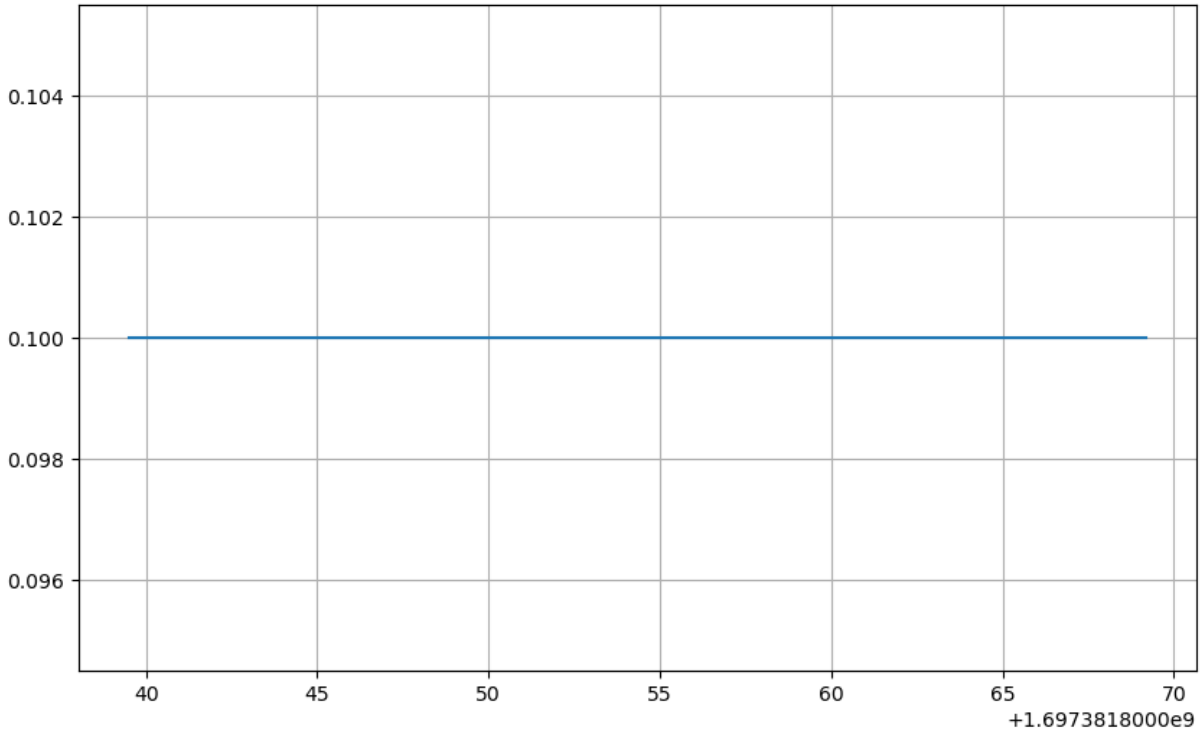
в vmstat, можно узнать ,что всего 5 процентов времени программа выполняет полезную работу

Memory

1) lockbus

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.

Для начала запустим 1 процесс `stress-ng --lockbus 1 --timeout 30` и посмотрим на его потребление памяти через top -b -n 1



Можно увидеть, что потребление памяти константное — 0.1 от общего числа, что логично из определния теста

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

При нескольких процессах — тоже самое

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
677313	egor	20	0	60124	18400	17664	R	99,7	0,1	0:06.18	stress-+
677314	egor	20	0	60124	18400	17664	R	99,7	0,1	0:06.18	stress-+
677312	egor	20	0	60124	18400	17664	R	99,4	0,1	0:06.17	stress-+

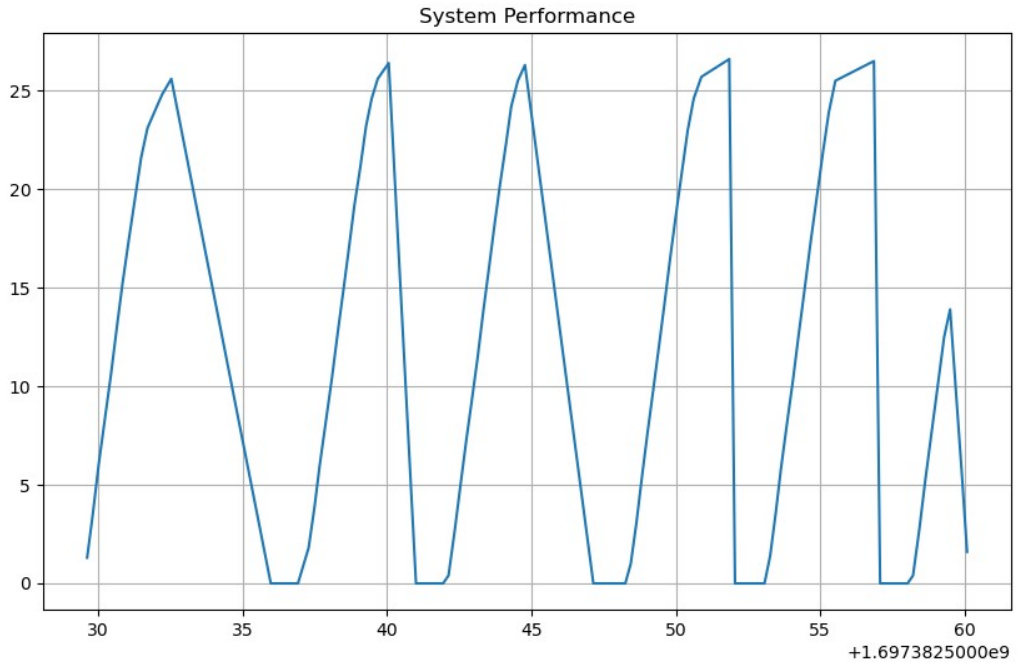
Каждый процесс использует одинаковое количество памяти

2) mmaphuge-maps

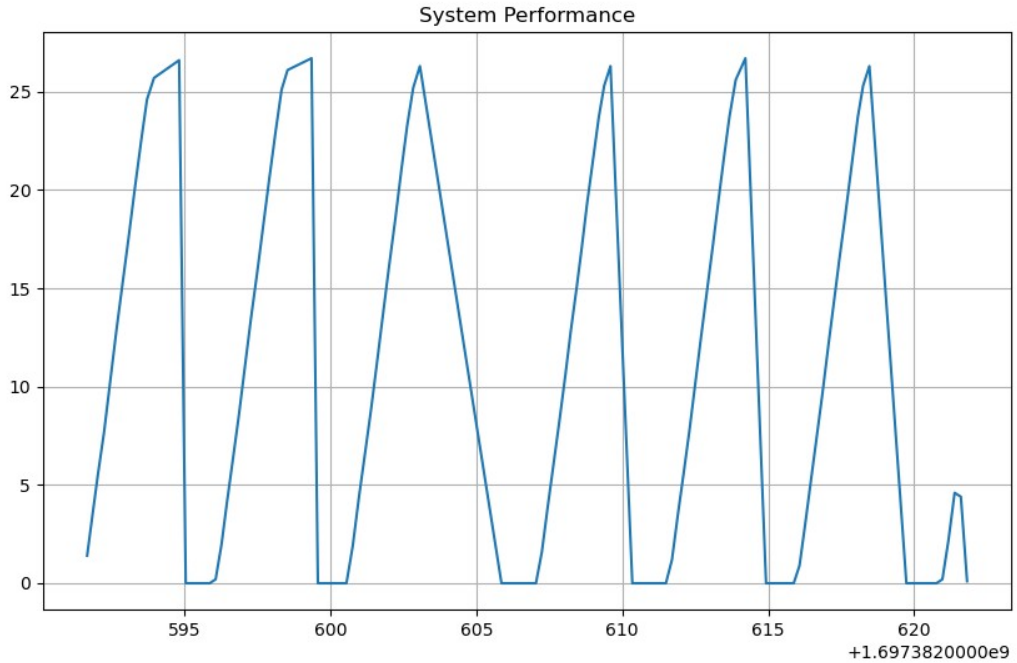
Устанавливает количество огромных маппингов, которое выполняется при тесте

Запускать будет командой
```
stress-ng --mmaphuge {i} --mmaphuge-mmaps {j} --timeout 30
```

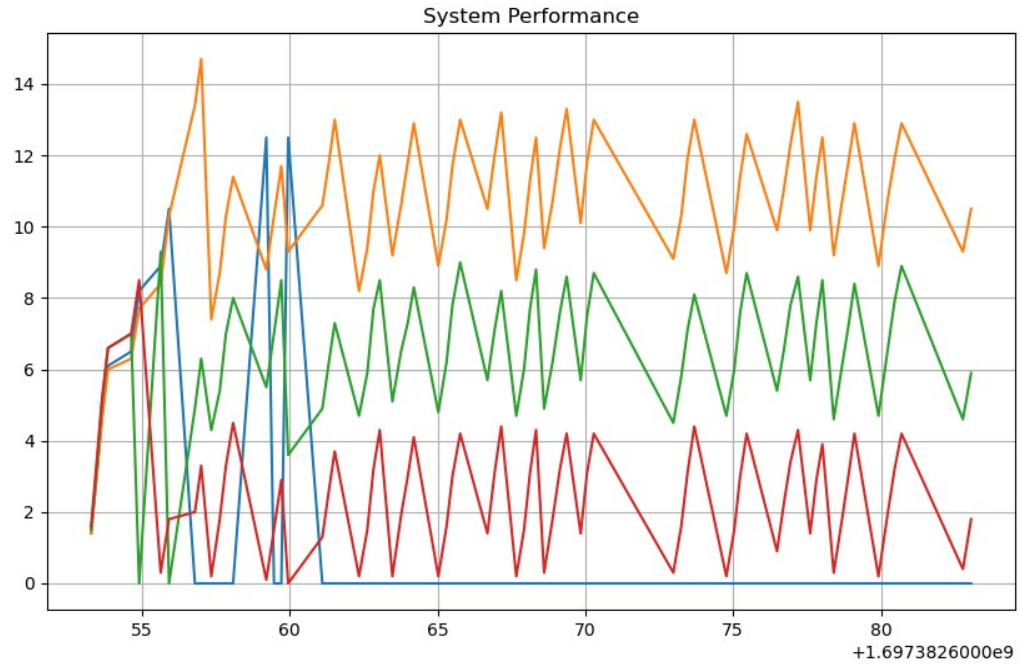
перебирая количество процессов, делающих огромные маппинги, и их количество
с помощью топа выведем использованием памяти каждым процессом
1 процесс 8129 маппингов



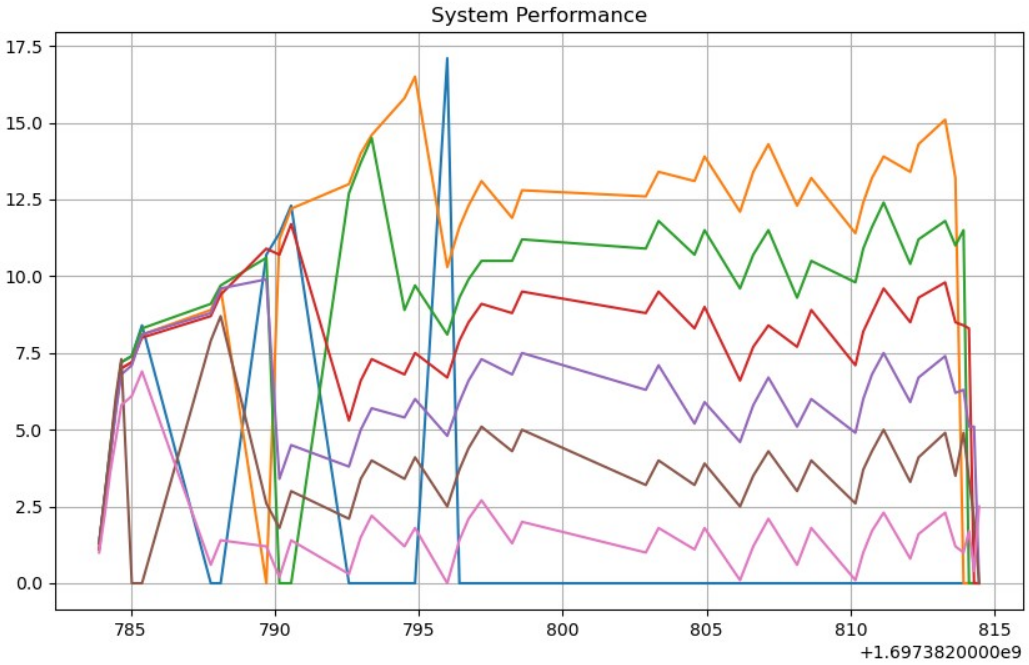
1 процесс 24576 маппингов



4 процесса 16386 маппингов



7 процессов 24576 маппингов



Network

1) Первый параметр

-- **sockdiag N** запускает N рабочих процессов, которые выполняют диагностику сетевых сокетов Linux sock_diag

Запустим тест командой stress-ng --sockdiag {i} --timeout 30

Для начала посмотрим доступные интерфейсы

```
root@OMEN-Laptop-15-en0xxx:/home/egor# nmcli dev status
DEVICE           TYPE      STATE                   CONNECTION
wlo1             wifi      подключено              Xiaomi_138
br-7b27a59f1036  bridge    подключено (внешнее)    br-7b27a59f1036
docker0          bridge    подключено (внешнее)    docker0
virbr0           bridge    подключено (внешнее)    virbr0
p2p-dev-wlo1     wifi-p2p  отключено               --
eno1             ethernet  недоступно              --
lo               loopback  не настроенно           --
```

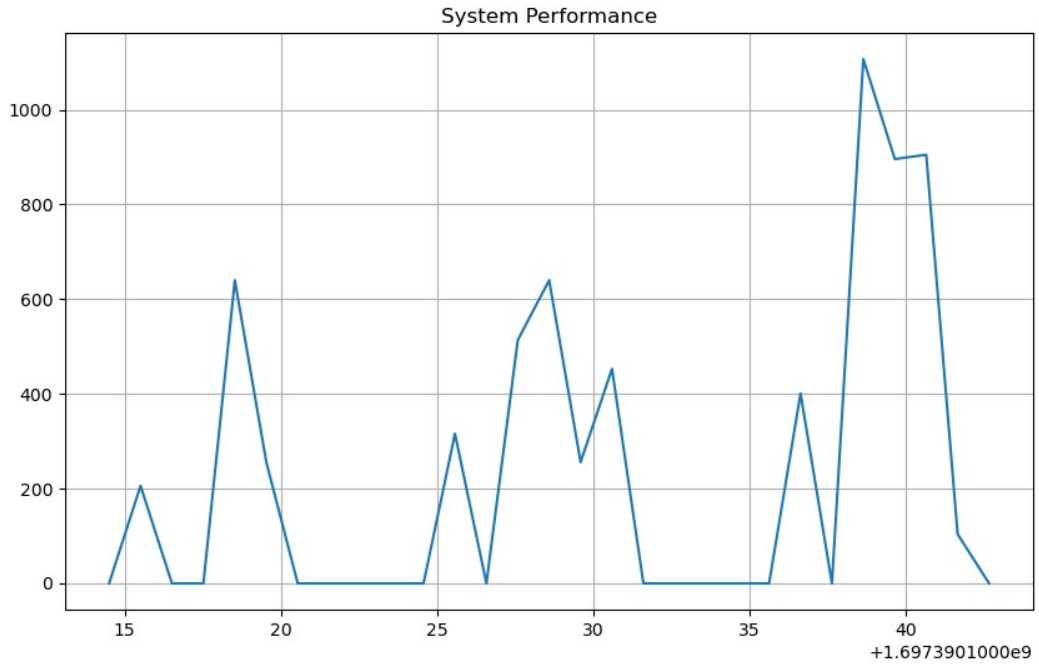
Сначала посмотрим, что данные действительно отправляются через iptraf

```
iptraf-ng 1.2.1
 Statistics for wlo1 ──────────────────────────────────────────────────────────

              Total       Total    Incoming    Incoming    Outgoing    Outgoing
            Packets       Bytes     Packets       Bytes     Packets       Bytes
Total:          106       11609          47        7027          59        4582
IPv4:           106       11609          47        7027          59        4582
IPv6:             0           0           0           0           0           0
TCP:            102       11321          45        6875          57        4446
UDP:              4         288           2         152           2         136
ICMP:             0           0           0           0           0           0
Other IP:         0           0           0           0           0           0
Non-IP:           0           0           0           0           0           0
Broadcast:        0           0           0           0           0           0

Total rates:          2,39 kbps          Broadcast rates:        0,00 kbps
                         4 pps                                     0 pps

Incoming rates:       1,08 kbps
                         2 pps
                                         IP checksum errors:          0
Outgoing rates:       1,31 kbps
                         2 pps
```

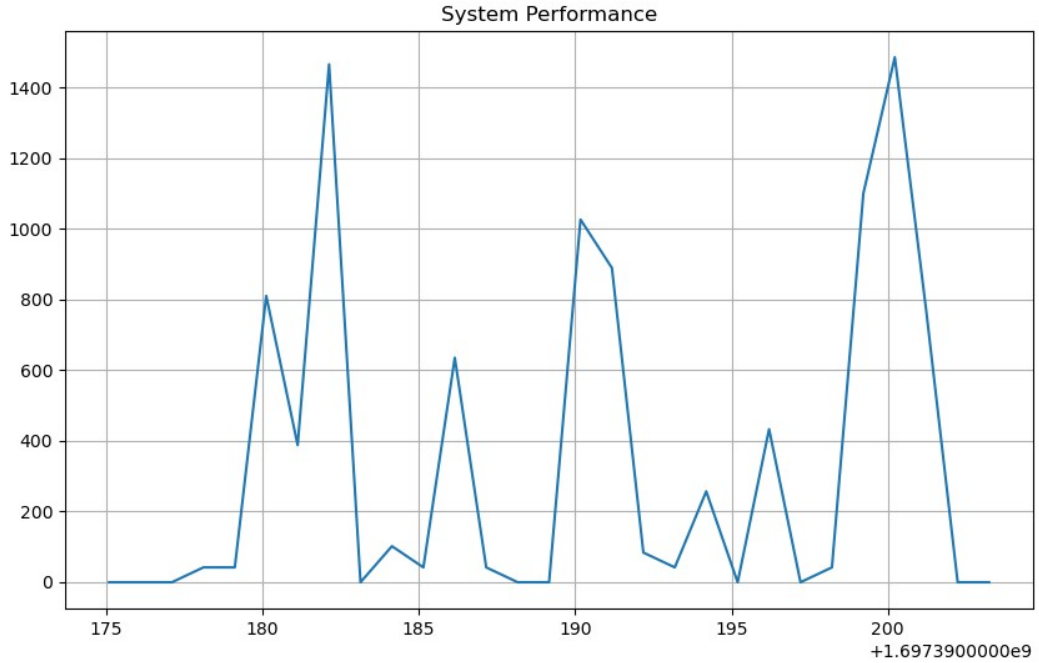
однако интерактивные инструменты не слишком хороши для построения графиков

С помощью ip -s link show wlo1(показывает статистику по интерфейсу wlo1-wireless interface) будем замерять скорость. Для этого будем искать разность полученных + отправленных байт каждую секунду

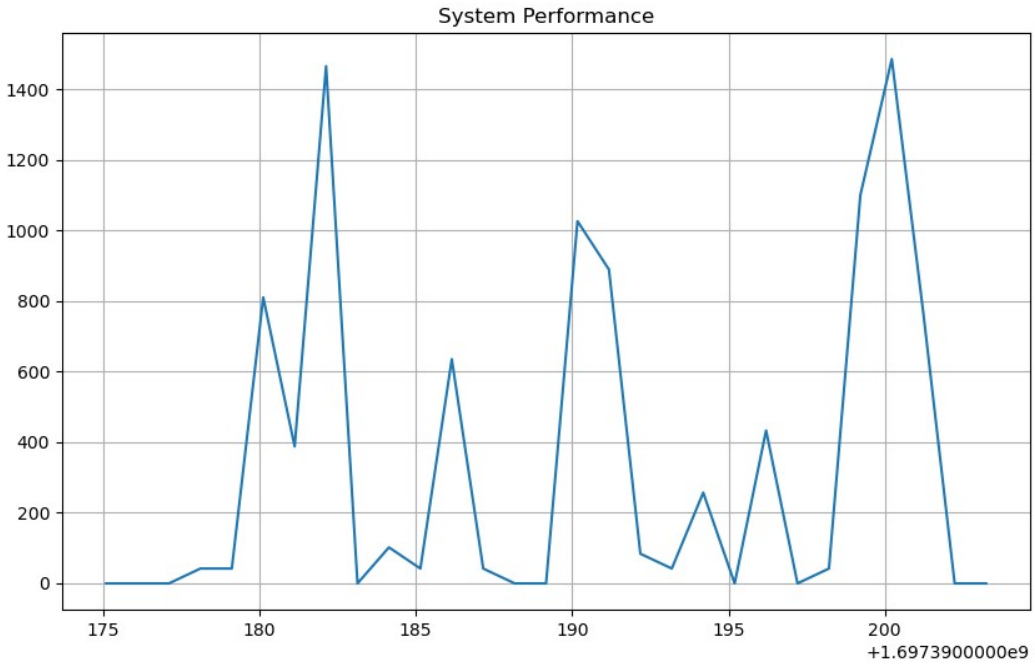
Скорость в байтах в секунду



3 процесса



5 процессов



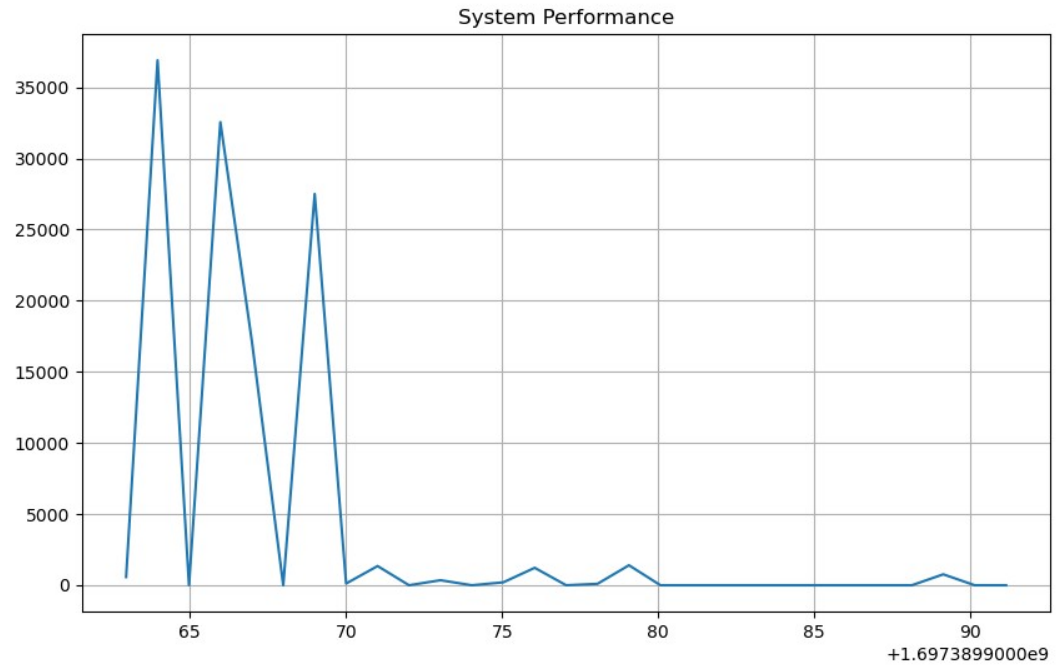
Видно, что скорость сильно меняется.

2) netlink-task

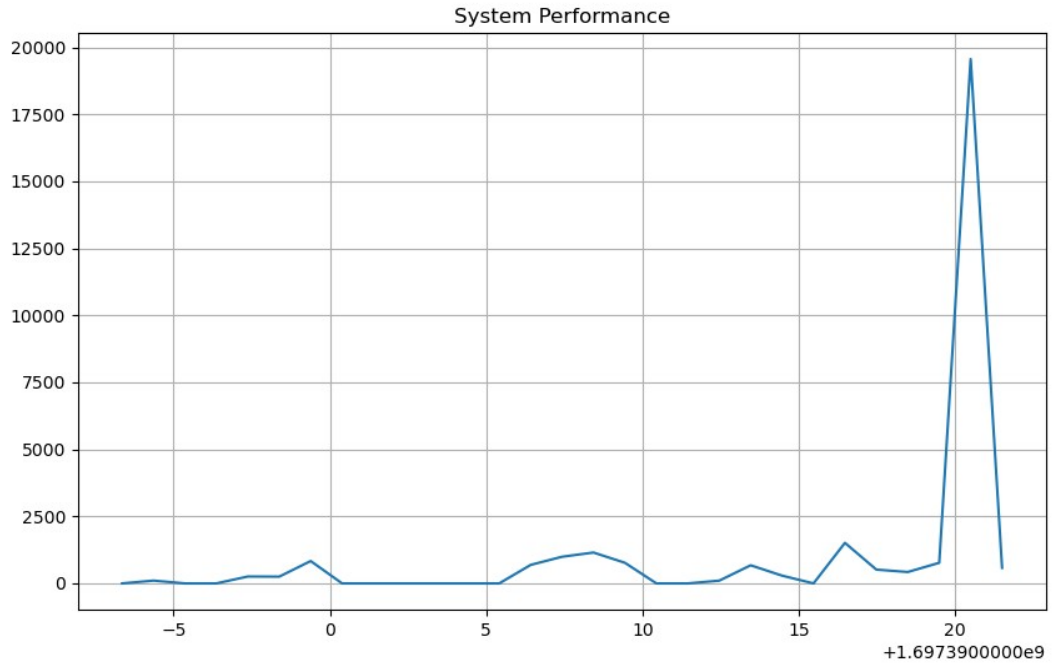
Запускает N рабочих процессов, которые собирают статистику через taskstats интерфейс.

Аналогично протестируем командой ip

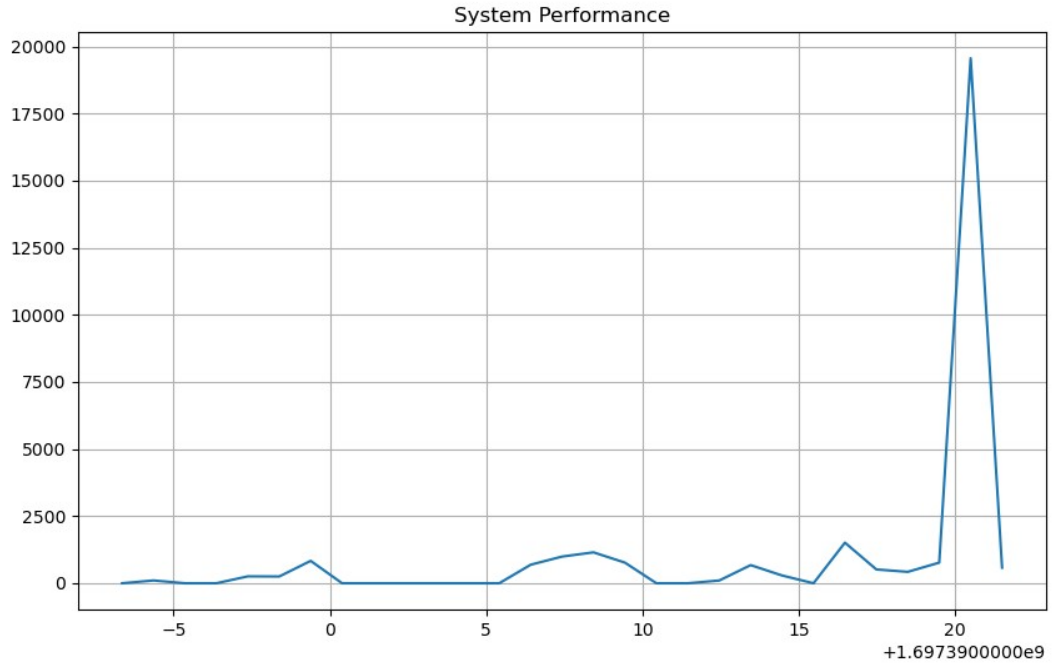
1 процесс



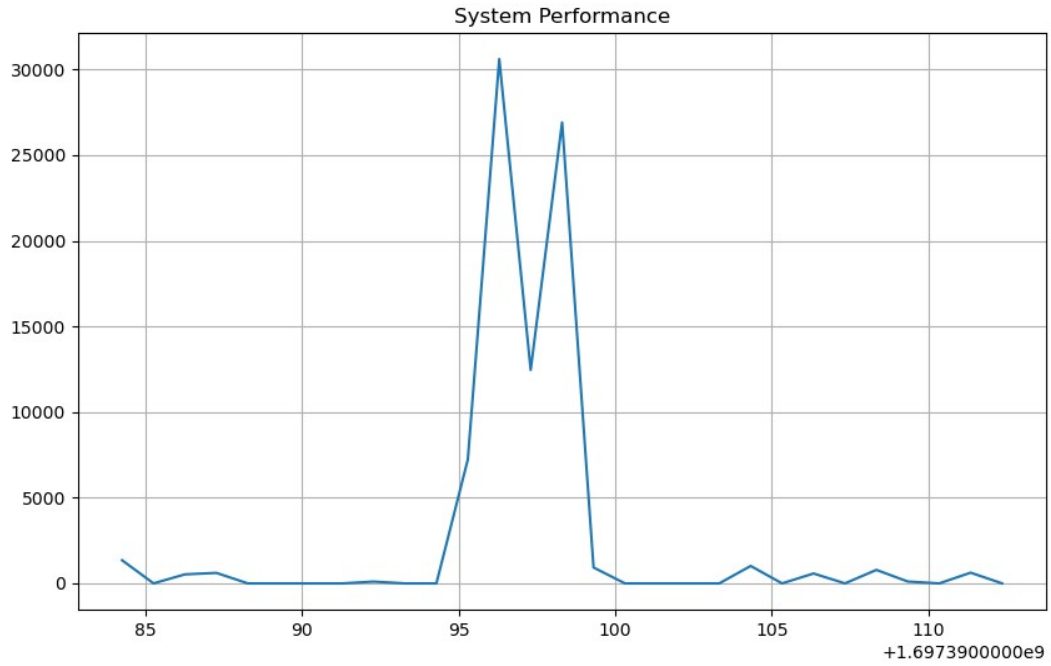
3 процесса



5



9 процессов



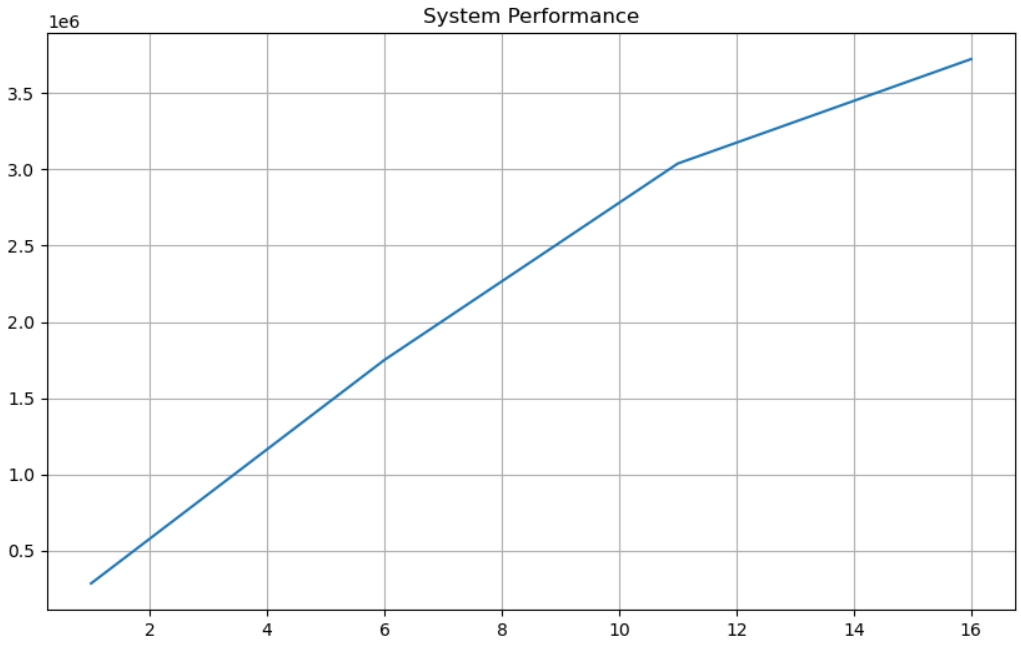
Наблюдается какая-то тенденция по снижению нагрузки на сетевую подсистему, которая, однако, не оправдывается из-за графика с 9 процессами.

Pipe

1) sigpipe

Запустить N рабочих процессов, которые неоднократно порождают дочерний процесс, который завершается до того, как родительский процесс завершает запись в канал, вызывая сигнал SIGPIPE

Пайп связывает 2 процесса для передачи данных. Убийства процессов будут вызывать переключения контекста, которые мы и будем мониторить утилитой perf.



Видно, что с ростом количества процессов растет и количество переключений контекста. Связать это с пайпом нельзя, так как и без этого параметра переключения контекстов увеличивались бы.

2) piperherd-yield

принудительно выполняет планирование после каждой записи, это увеличивает скорость переключения контекста.

В этот раз не будем строить график, а сделаем все руками
Сравним количество переключений контекста без этого параметра и с ним

```
 Performance counter stats for 'stress-ng --pipe 1 --pipeherd-yield --timeout 30s':

          121 454      context-switches

      30,021794320 seconds time elapsed

      12,602766000 seconds user
      46,910210000 seconds sys


root@OMEN-Laptop-15-en0xxx:/home/egor/Рабочий стол# sudo perf stat -e context-switches stress-ng -
stress-ng: info:  [1804448] setting to a 30 second run per stressor
stress-ng: info:  [1804448] dispatching hogs: 1 pipe
stress-ng: info:  [1804448] successful run completed in 30.00s

 Performance counter stats for 'stress-ng --pipe 1 --timeout 30s':

           72 174      context-switches

      30,023150307 seconds time elapsed

      12,832707000 seconds user
      46,893386000 seconds sys
```

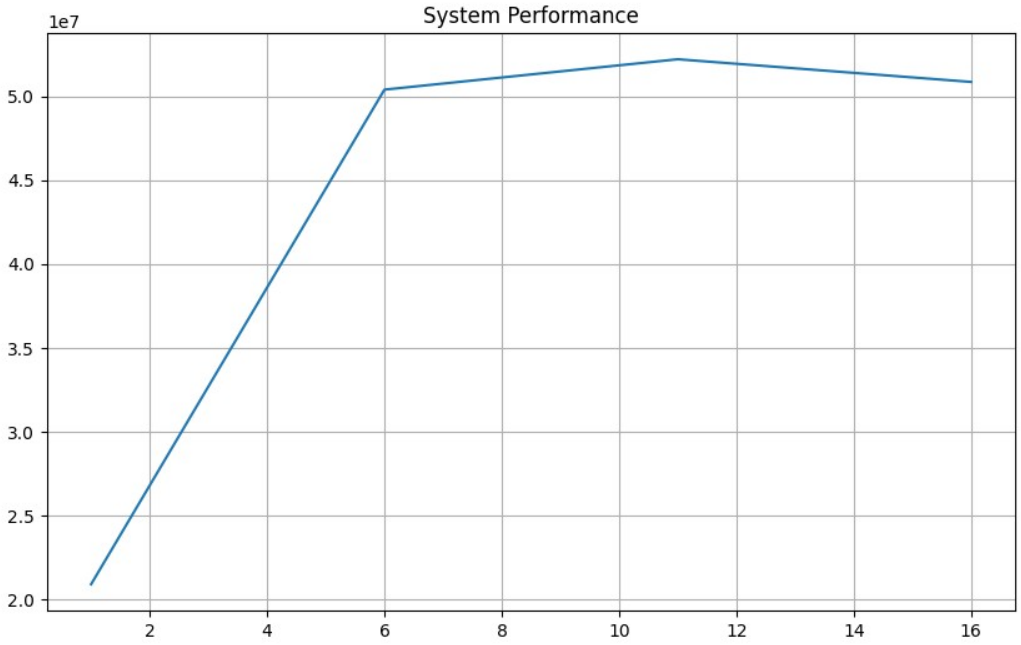
Видно, что без параметра меньше

Sched

1) reshed

Запустить N процессов, которые осуществляют перепланирование процесса. Каждый стрессор порождает дочерний процесс для каждого из положительных nice priority и перебирает nice priority от 0 до уровня самого низкого приоритета (самого высокого значения приятности). Для каждого из хороших уровней 1024 итерации более 3 политик планирования не в реальном времени и возникает sched_yield, вызывающий интенсивную деятельность по перепланированию.

Промониторим переключения контекста, как мы делали ранее(перфом)

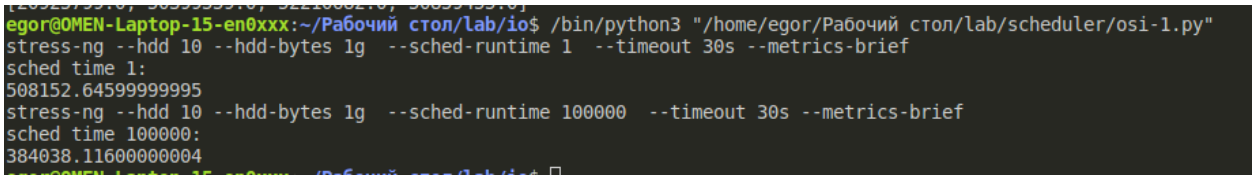


В mpstat видно, что большую часть времени процесс занят не пользовательской задачей

```
egor@OMEN-Laptop-15-en0xxx:~$ mpstat
Linux 6.2.0-34-generic (OMEN-Laptop-15-en0xxx)  18.10.2023      _x86_64_       (16 CPU)

01:08:53     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
01:08:53     all    5,62    0,09    4,24    2,35    0,00    0,33    0,00    0,00    0,00   87,37
```

2) sched-runtime

Устанавливает максимальное время для deadline scheduler
deadline scheduler — это io scheduler. Поэтому мы будем смотреть на то, как много операций записи будет проходить с разными значениям этого флага
сначала установис 100000 нс, а затем 1.С помощью iostat получим данные о скорости записи на диск, и сравним



Средняя скорость записи при маленьком значении флага заметно выше

Это может быть связано с тем, что долгие операции ввода вывода сразу откидываются.

Вывод:

В ходе работы я ознакомился с различными утилитам системного анализа и мониторинга и применил их на практике.