Национальный исследовательский университет ИТМО

Лабораторная работа №1

«Операционные системы»

Выполнил:

студент группы P33111

Чесноков А. А.

Преподаватель:

Осипов С. В.

Задание

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 рассмотрены 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).

Бестиарий:

1) BOGOOPS (Bogus Operations Per Second) – 1 циклическая итерация действия нагрузчика.
2) FLOPS (Floating-point Operations Per Second) – количество итераций с плавающей запятой за 1 секунду.
3) Read/Write Speed – думаю, понятно.
4) Network Speed – думаю, тоже.

Вариант:

cpu: [clongdouble, psi];

cache: [l1cache-line-size, l1cache-ways];

io: [iomix, io-uring];

memory: [prefetch, memthrash];

network: [netlink-proc, dccp];

pipe: [pipeherd-yield, pipeherd];

sched: [sched-period, resched]

Построить графики (подходящие по заданию.):

- Потребления программой CPU;
- Нагрузки, генерируемой программой на подсистему ввода-вывода;
- Нагрузки, генерируемой программой на сетевую подсистему;
- Другие графики, необходимые для демонстрации работы.

Лаборатория:

Для начала посмотрим, что из себя представляет мой процессор

```
~ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  8
  On-line CPU(s) list:   0-7
Vendor ID:               GenuineIntel
  Model name:            11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
    CPU family:          6
    Model:               140
    Thread(s) per core:  2
    Core(s) per socket:  4
    Socket(s):           1
    Stepping:            1
    CPU(s) scaling MHz:  24%
    CPU max MHz:         4700,0000
    CPU min MHz:         400,0000
    BogoMIPS:            5608,00
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 c
                         rt arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfm
                         pr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsa
                         ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid
                         adx smap avx512ifma clflushopt clwb intel_pt avx512cd sha_ni avx512bw avx
                         ify hwp_act_window hwp_epp hwp_pkg_req avx512vbmi umip pku ospke avx512_v
                         4b fsrm avx512_vp2intersect md_clear flush_l1d arch_capabilities
Virtualization features:
  Virtualization:        VT-x
Caches (sum of all):
  L1d:                   192 KiB (4 instances)
  L1i:                   128 KiB (4 instances)
  L2:                    5 MiB (4 instances)
  L3:                    12 MiB (1 instance)
NUMA:
  NUMA node(s):          1
  NUMA node0 CPU(s):     0-7
Vulnerabilities:
  Gather data sampling:  Mitigation; Microcode
  Itlb multihit:         Not affected
  L1tf:                  Not affected
  Mds:                   Not affected
  Meltdown:              Not affected
  Mmio stale data:       Not affected
```

Параметры следующие: **clongdouble и psi**. Далее следуют определения, полученные из man по stress-ng: cloungdouble – 1000 повторений сложных операций с миксованием чисел с плавающей точкой типа long double; psi – вычисление φ (обратная константа Фибоначчи), используя сумму обратных чисел Фибоначчи.

Clongdouble

Запустим для начала 1 процесс

```
~ stress-ng --metrics --cpu-method clongdouble --cpu 1
                                                                                ✔
stress-ng: info:  [57178] defaulting to a 1 day, 0 secs run per stressor
stress-ng: info:  [57178] dispatching hogs: 1 cpu
stress-ng: info:  [57178] note: 8 cpus have scaling governors set to powersave and this can impact on perfo
rmance; setting /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor to 'performance' will improve perform
ance
```

Видим сообщение об использовании scaling governors. Вкратце: операционная система может управлять частотой процессора для улучшения производетельности или в целях экономии заряда батареи. Ядро осуществляет эти злодеяния через подсистему CPUFreq, котороая предоставляет 2 уровня абстракции: scaling governors и scaling drivers. Первые отвечают за алгоритмы, по которым расчитывается выходная частота процессора, основанная на нуждах системы. Вторые отвечают за непосредственное взаимодействие с ЦПУ.

Посмотрим, что есть в файле sys/devices/system/cpu/cpu*/cpufreq/scaling_governor

```
~ cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
powersave
powersave
powersave
powersave
powersave
powersave
powersave
powersave
```

Меняем все на performace и запускаем команду снова.

```
~ stress-ng --metrics --cpu-method clongdouble --cpu 1                          ✔ 2m 10s
stress-ng: info:  [57945] defaulting to a 1 day, 0 secs run per stressor
stress-ng: info:  [57945] dispatching hogs: 1 cpu
^Cstress-ng: metrc: [57945] stressor       bogo ops real time  usr time  sys time   bogo ops/s     bogo ops
/s CPU used per       RSS Max
stress-ng: metrc: [57945]                           (secs)    (secs)    (secs)   (real time) (usr+sys time)
 instance (%)          (KB)
stress-ng: metrc: [57945] cpu             147661     88.29     88.18      0.00      1672.40        1674.49
       99.87          1260
```

Воспользуемся командой pidstat, чтобы посмотреть на потребление cpu процессом stress-ng

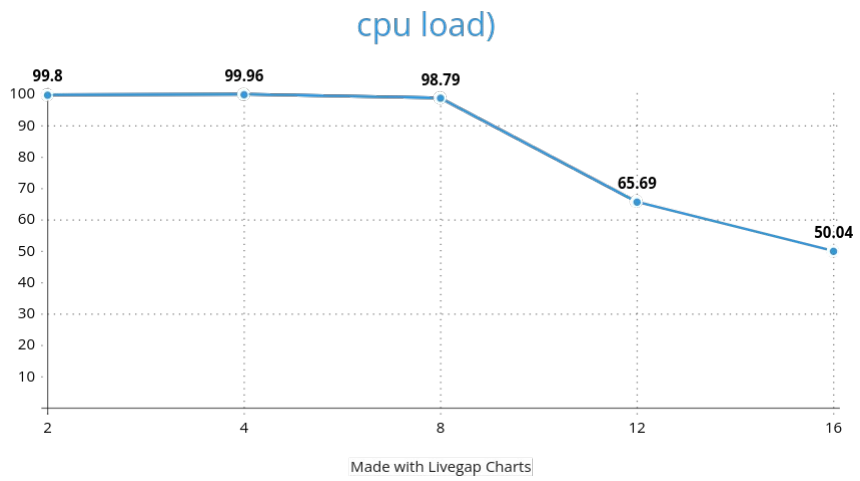
```
~ pidstat -p 57946 1
Linux 5.10.202-1-MANJARO (cleanyco)     05.12.2023      _x86_64_        (8 CPU)

18:18:33      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
18:18:34     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:35     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:36     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:37     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:38     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:39     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:40     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:41     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:42     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:43     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:44     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:45     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
18:18:46     1000     57946  100,00    0,00    0,00    0,00  100,00     5  stress-ng-cpu
```

Ядро загружено полностью.

Сделаем тесты раз в 5 секунд с разным количеством одновременно работающих стрессоров. [2, 4, 8, 12, 16]

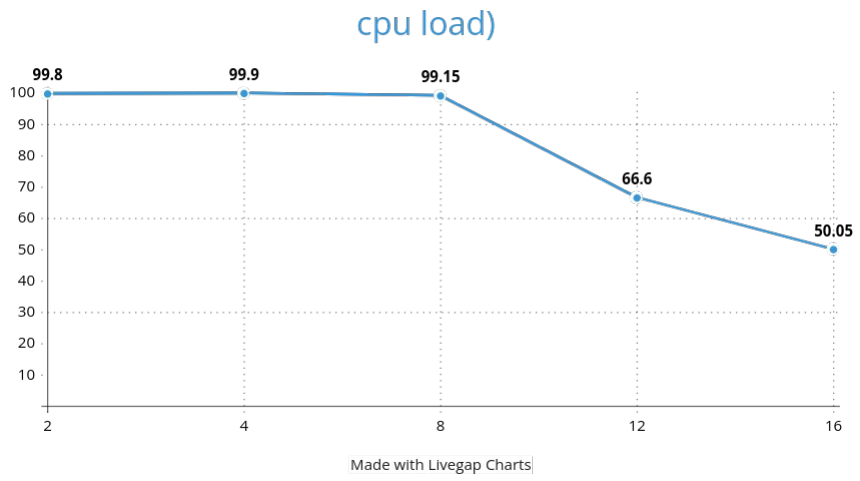
По горизонтали указано количество стрессоров, по вертикали — средняя нагрузка на 1 ядро процессора.



Результаты очевидны.

PSI

Сделаем сразу тесты на [2, 4, 8, 12, 16] стрессорах.



Результаты получились почти идентичными.

Cache

l1cache-line-size

Кеш-линия: ячейчка записи, куда вы можете сохранить кусочек из основной памяти.

Как будем тестировать: использовать будем в паре с флагом **—l1cache N,** который будет отвечать за количество стрессоров, которые будут нагружать кеши чтением и запись. Будем (сколько раз я уже повторил это слово) изменять объем кеш-линий первого уровня и смотреть на количество промахов. Анализ проводим командой **perf**.

Команда: **perf stat -e L1-dcache-load-misses stress-ng --l1cache {i} --l1cache-line-size {j} -t 5s**

С помощью этого метода нагрузки мы проверяем производительность кэшей процессора.

```
perf stat -e L1-dcache-load-misses stress-ng --l1cache 2 --l1cache-line-size 2 -t 5s
stress-ng: info:  [18989] setting to a 5 secs run per stressor
stress-ng: info:  [18989] dispatching hogs: 2 l1cache
stress-ng: info:  [18990] l1cache: l1cache: size: 48.0K, sets: 2048, ways: 12, line size: 2 bytes
stress-ng: info:  [18989] skipped: 0
stress-ng: info:  [18989] passed: 2: l1cache (2)
stress-ng: info:  [18989] failed: 0
stress-ng: info:  [18989] metrics untrustworthy: 0
stress-ng: info:  [18989] successful run completed in 5.82 secs

 Performance counter stats for 'stress-ng --l1cache 2 --l1cache-line-size 2 -t 5s':

     6 142 772 605      L1-dcache-load-misses:u
```

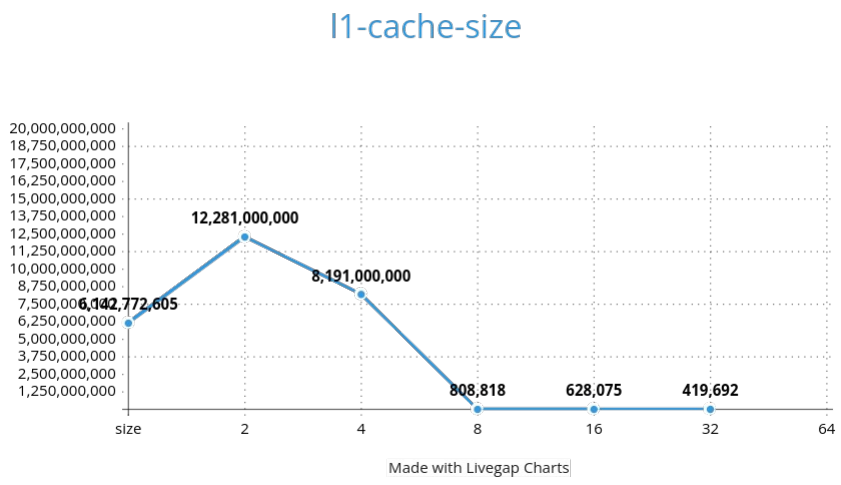
2 стрессора + 2 кб на кеш-линию = 6 миллиардов промахов. Результат: поезд сделал бум. Очевидно, что если увеличит кол-во воркеров, кол-во промахов увеличится.

```
perf stat -e L1-dcache-load-misses stress-ng --l1cache 8 --l1cache-line-size 2 -t 5s
stress-ng: info:  [19141] setting to a 5 secs run per stressor
stress-ng: info:  [19141] dispatching hogs: 8 l1cache
stress-ng: info:  [19142] l1cache: l1cache: size: 48.0K, sets: 2048, ways: 12, line size: 2 bytes
stress-ng: info:  [19141] skipped: 0
stress-ng: info:  [19141] passed: 8: l1cache (8)
stress-ng: info:  [19141] failed: 0
stress-ng: info:  [19141] metrics untrustworthy: 0
stress-ng: info:  [19141] successful run completed in 8.53 secs

 Performance counter stats for 'stress-ng --l1cache 8 --l1cache-line-size 2 -t 5s':

    24 575 710 041      L1-dcache-load-misses:u
```

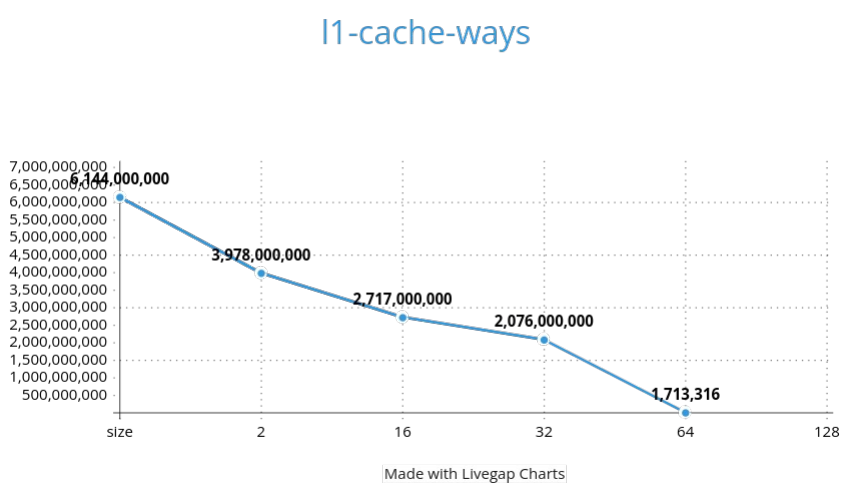
Оставляем 2 стрессора, но увеличиваем размер кеш-линии. Построим график.



Постепенно увеличивая размер кеш-линии до 64 байт, мы достигли размера кеш-линии современных процессоров (хотя там может быть от 32 до 128 байт). Оптимальный размер кеш-линии зависит от данных, с которыми работает программа. Меньший размер данных => меньший размер кеш-линии и наоборот.

l1cache-ways

Кеш-путь определяет количество кеш-линий, т.е. сколько кеш-линий может храниться одновременно. Повторим предыдущий эксперимент: оставляем 2 стрессора, меняем размер кэш-пути.



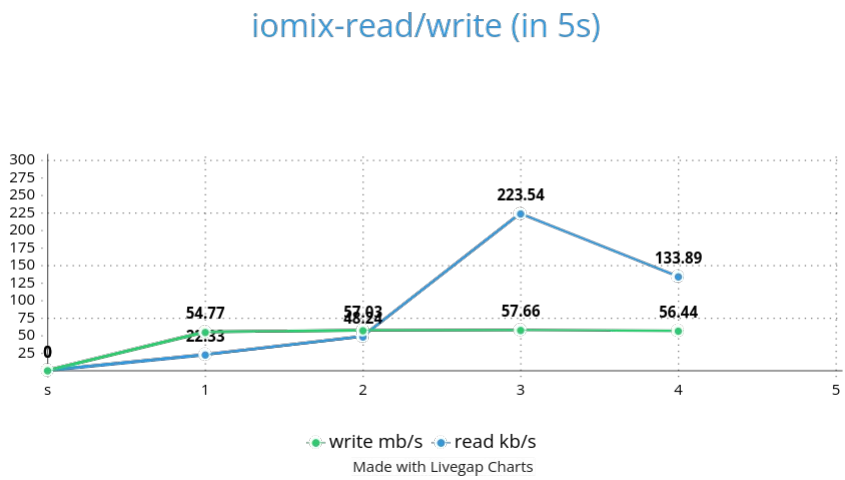
IO

Используем **iomix и io-uring** для создания нагрузки на подсистему ввода-вывода.

iomix

Данная команда порождает потоки, которые занимаются чтением, копированием, записью.

Команда: **stress-ng --iomix 16 -t 5s --metrics | sudo iotop -b -o -n 5**



Построили график по времени за 5 секунд. Очевидно, что при увеличении числа воркеров будет падать скорость чтения и записи, но здесь другой случай. Оставили 16 воркеров и смотрим на скорость чтения и записи. После 5 секунд скорость чтения и записи устаканивается и почти не изменяется (поверьте наслово). Предположу, что это связано с постепенным перераспределением ресурсов в сторону системы ввода-вывода.

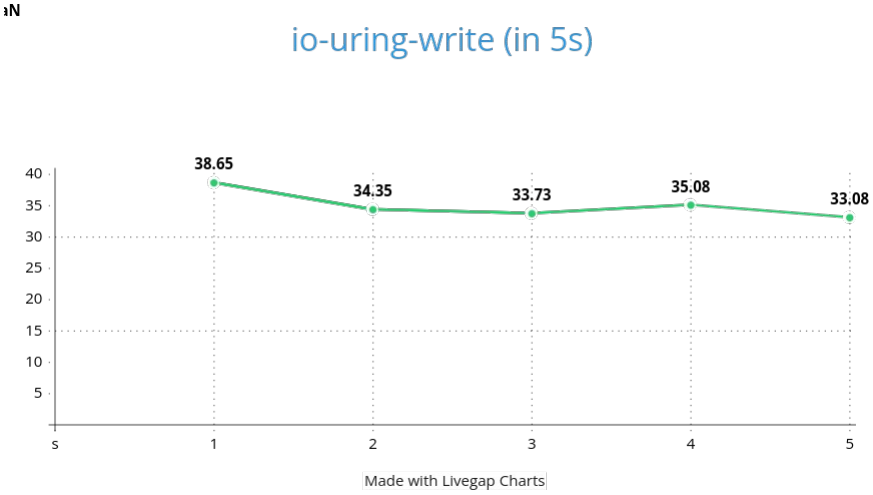
io-uring

Тестирование системы io с помощью механизма io_uring. Нагрузка в виде чтения, записи, открытия и закрытия файлов. На каждый цикл 1024x512 байт пишутся во временный файл и читаются из него же.

io-uring это интерфейс системных вызовов ядра Linux для асинхронных операций ввода-вывода.

Команда: **stress-ng --io-uring 2 -t 5s --metrics | sudo iotop -b -o -n 5**

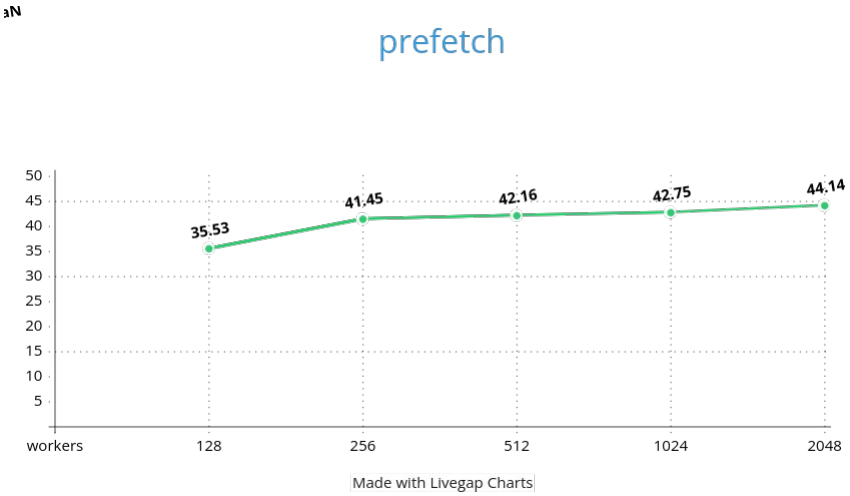
Тут начинаются некоторые проблемы. Даже при запуске с правами супер пользователя, io-uring, не производит чтение, только запись. Предположу, что это баг, потому что предыдущий метод отлично и читал, и писал. Посмотрим на график записи за 5 секунд при 2-х воркерах.



Memory

prefetch

Запускает N воркеров, которые тестируют prefetch и non-prefetch чтение буфера, размером с L3 cache.



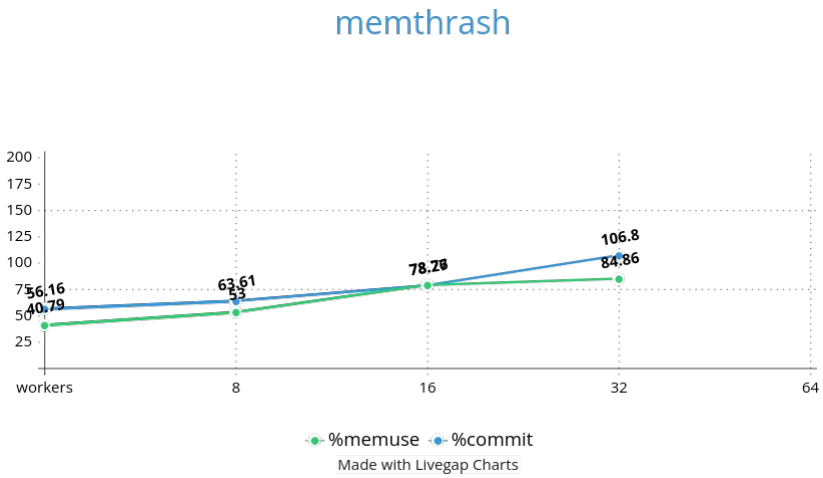
Ничего удивительного, в следствие малой загрузки командой памяти, посмотрим на вывод команды и тоже скажем, что ничего страшного не произошло.

kbmemfree	kbavail	kbmemused	%memused	kbbuffers	kbcached	kbcommit	%commit	kbactive	kbinact	kbdirty
4615004	7906112	5984980	37,28	774780	4150148	16395112	53,40	6601432	2724440	372
4013516	7304632	6587072	41,03	774780	4146604	17410216	56,71	7190704	2724440	372
3415912	6707088	7182164	44,74	774780	4146156	18488364	60,22	7773184	2724440	372
2950944	6242208	7644688	47,62	774780	4145904	19263224	62,74	8228104	2724440	372
2540328	5831672	8033124	50,04	774780	4166384	19854908	64,67	8616928	2724440	372
3507141	6798342	7086406	44,14	774780	4151039	18282365	59,55	7682070	2724440	372

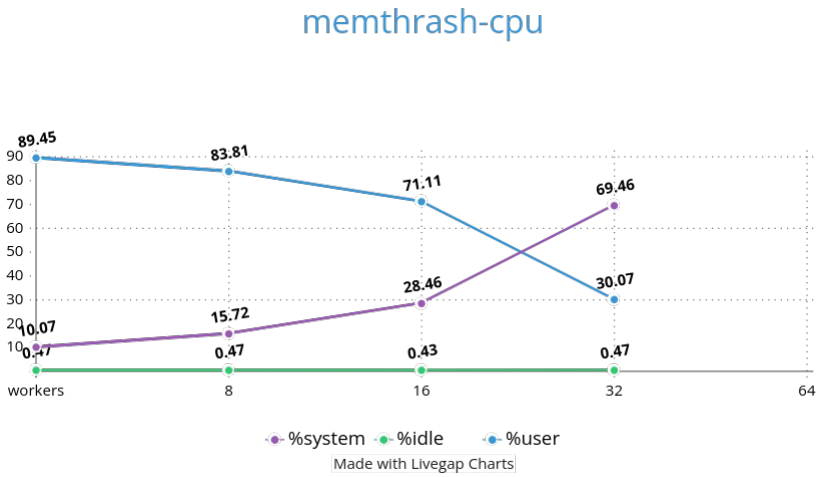
Memthrash

Запускает N воркеров, которые будут грузить буфер размером 16 мб, чтобы спровоцировать отключение по перегреву (?!). Каждый стрессор запускает как минимум 1 поток на cpu. Оптимальным выбором для N является значение, которое делится на количество ядер в вашем процессоре.

D



Можно заметить, что %commit (процент памяти, необходимый для текущей нагрузки RAM + SWAP) перевалил за 100. Так бывает, когда система использует больше памяти, чем есть.



Ну и нагрузку на CPU тоже можно посмотреть.

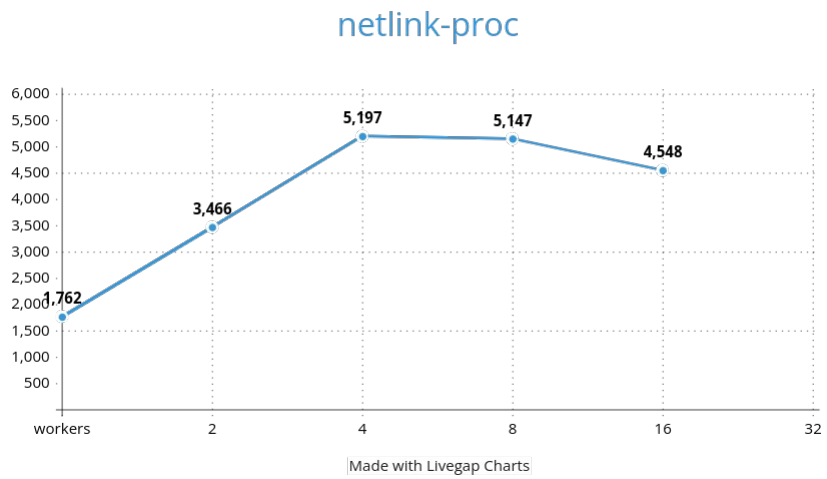
Network

netlink-proc: start N workers that spawn child processes and monitor fork/exec/exit process

events via the proc netlink connector. Each event received is counted as a bogo op.

Что такое **netlink** в Linux? Это способ коммуникации между ядром линукса и юзерспейсом, а именно взаимодействия процессов пользователя и процессов ядра.

В качестве измеряемой единицы будем отслеживать количество **bogops** (выше написано, что each event received…). Запустим с правами sudo: **sudo stress-ng –netlink-proc {I} –metrics -t 5s.**

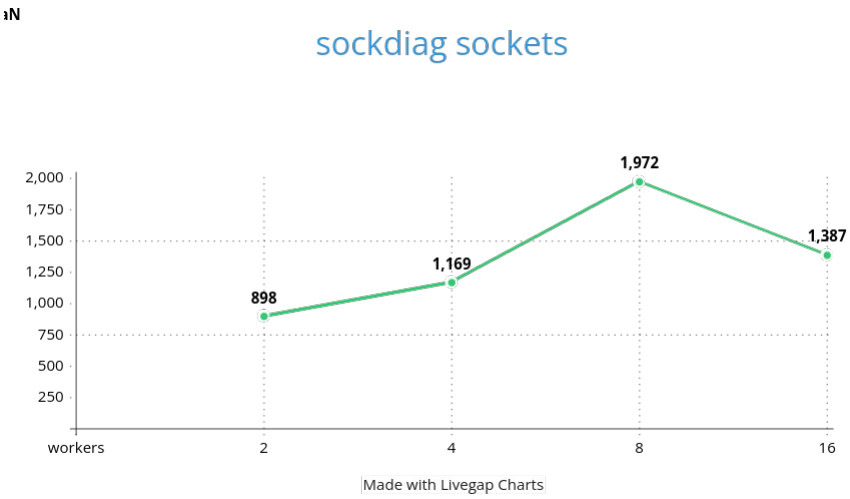


Результат: поезд снова сделал бум.

dccp: на моей машине не поддерживается, разрешили заменить на что захочу. Захотел на **sockdiag.**

Эта приблуда выполняет диагностику сокетов, используя подсистему sock_diag. Она позволяет получать информацию о состоянии сокетов (открытые соединения, прослушиваемые порты и т.д.)

В этот раз запустим следующую команду: **sudo stress-ng --sockdiag {i} --metrics -t 5s | sar 1 10 -n SOCK (для подсчета активных сокетов) / DEV (для просмотра статистики по сетевым операциям).** Ключ -n отвечает за статистику по network, sock это статистика по сокетам (сокеты это механизм для передачи данных между двумя процессами). totsck – количество сокетов, используемых в системе.



Тут надо сделать пометку, что учитываются все сокеты, то есть не только те, которые создаются stress-ng. Поэтому мы берем среднее число. На первых секундах количество сокетов наибольшее, затем оно начинает постепенно снижаться.

Но что если посмотреть через ключ **DEV**?

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcst/s	%ifutil
lo	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
wlo1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcst/s	%ifutil
lo	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
wlo1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcst/s	%ifutil
lo	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
wlo1	**1,00**	**1,00**	**0,17**	**0,08**	0,00	0,00	0,00	0,00

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcst/s	%ifutil
lo	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
wlo1	**1,00**	**1,00**	**0,17**	**0,08**	0,00	0,00	0,00	0,00

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcst/s	%ifutil
lo	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
wlo1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcst/s	%ifutil
lo	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
wlo1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcst/s	%ifutil
lo	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
wlo1	**17,00**	**21,00**	**4,49**	**3,08**	0,00	0,00	0,00	0,00

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcst/s	%ifutil
lo	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
wlo1	0,00	**2,00**	0,00	**0,18**	0,00	0,00	0,00	0,00

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcst/s	%ifutil
lo	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
wlo1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcst/s	%ifutil
lo	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
wlo1	**1,90**	**2,50**	**0,48**	**0,34**	0,00	0,00	0,00	0,00

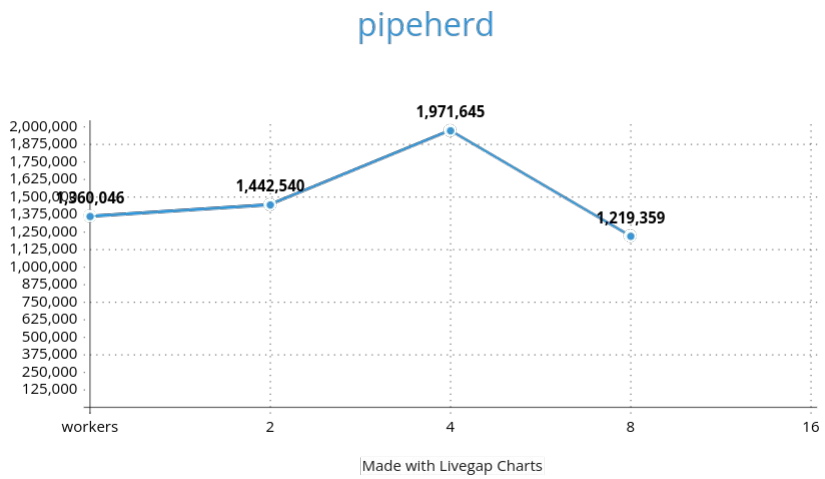
rxpck/s — total количество пакетов, полученных за секунду. txpck/s — отправленных за секунду.

Остальное это КБ полученные и отправленные за секунду. 2.5 кб и 0.48 кб? Количество стрессоров не повиляло на передачу пакетов. Будем считать, что так и должно быть :(

Pipe

pipeherd: создает каждому воркеру по 100 дочерних процессов, которые будут обмениваться данными через общие каналы. Это вызывает быструю смену контекста, которая может застопорить процессы: процессы пробуждаются, находятся в состоянии готовности, но не попадают на процессор. Замеряем количество смен контекста.

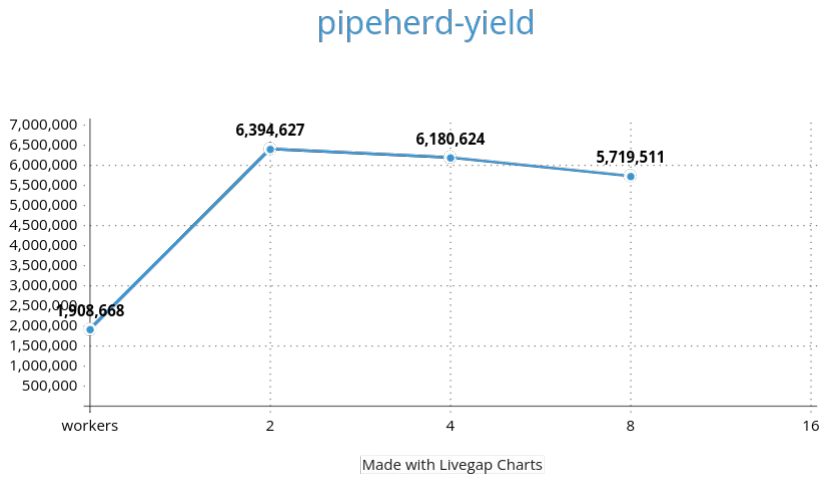
Команда: **sudo perf stat -e cs sudo stress-ng --pipeherd 32 -t 5s --metrics**



Видим, что число смен контекста уменьшилось. Имхо, это связано с планировщиком и с тем, как он раздает приоритеты этим процессам. Возможно, не все процессы успевают получить равное процессорное время.

Pipeherd-yield: принудительный вызов планировщика после каждой записи, это увиличивает скорость переключения контекста (если судить по man). Проверим.

Команда: **sudo perf stat -e cs sudo stress-ng --pipeherd --pipeherd-yield 32 -t 5s —metrics**

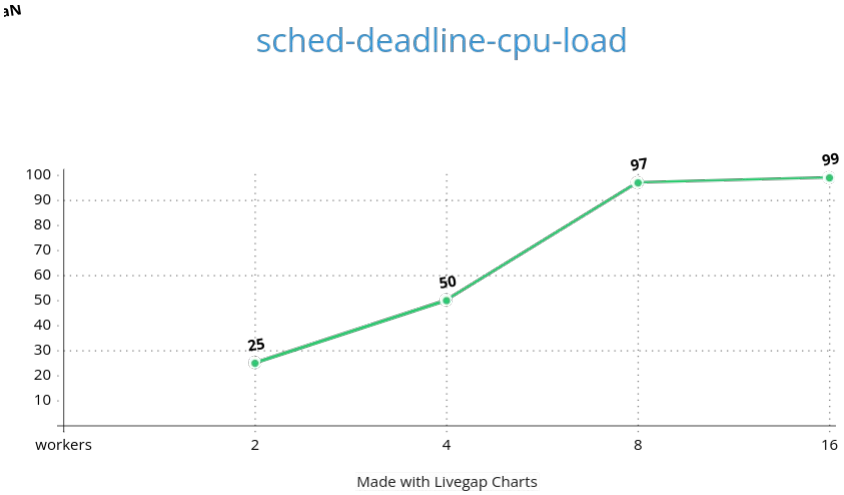


SCHED

sched-period

Работаем с планировщиком процессов. Команда позволяет установить период для deadline-scheduler. Значение по умолчанию 0 (в наносекундах). Для чего нужен deadline-scheduler? Он гарантирует время запуска для какого-то запроса. Он влияет на все операции ввода-вывода. Отвечает за две очереди операций: чтения и записи. Очереди отсортированы по деадлайну. Приоритеты отдаются очереди чтения, потому что при чтении, зачастую, процессы блокируются.

Команда: **sudo stress-ng --cpu {i} --sched-period 2 -t 5s --metrics | sar 1 10.** Будем менять количество воркеров при дедлайне в 2 нс.



Более того, у нас крайне низкий процент простоя.

```
~ sudo stress-ng --cpu 16 --sched-period 5 -t 5s --metrics | sar 1 10
Linux 6.1.66-1-MANJARO (cleanyco)       11.12.2023      _x86_64_        (8 CPU)

12:16:14        CPU     %user     %nice   %system   %iowait    %steal     %idle
12:16:15        all     97,74      0,00      0,38      0,00      0,00      1,88
12:16:16        all     99,75      0,00      0,25      0,00      0,00      0,00
12:16:17        all     99,75      0,00      0,25      0,00      0,00      0,00
12:16:18        all     99,62      0,00      0,38      0,00      0,00      0,00
12:16:19        all     99,88      0,00      0,12      0,00      0,00      0,00
12:16:20        all      6,48      0,00      0,25      0,25      0,00     93,02
12:16:21        all      0,50      0,00      0,25      0,00      0,00     99,25
12:16:22        all      0,38      0,00      0,25      0,00      0,00     99,37
12:16:23        all      0,50      0,00      0,00      0,00      0,00     99,50
12:16:24        all     15,90      0,00      2,24      0,00      0,00     81,86
Average:        all     52,02      0,00      0,44      0,03      0,00     47,52
```

resched

Запускает N воркеров, которые запускают «перепланирование». Каждый стрессор спавнит child-процесс процессу с положительным уровнем nice и итерируется по положительным уровням nice (от 0 до самого большего — меньшего по приоритету). Для каждого nice-уроня итерируется 1024 раз.

Команда: **sudo stress-ng --resched {i} -t 5s --metrics | sar 1 10**

```
~ sudo stress-ng --resched 8 -t 5s --metrics | sar 1 10
Linux 6.1.66-1-MANJARO (cleanyco)       11.12.2023      _x86_64_        (8 CPU)

13:05:14        CPU     %user     %nice   %system   %iowait    %steal     %idle
13:05:15        all      6,62      4,75     86,50      0,00      0,00      2,12
13:05:16        all      1,13      8,77     90,10      0,00      0,00      0,00
13:05:17        all      1,37      7,99     90,64      0,00      0,00      0,00
13:05:18        all      1,75      8,35     89,90      0,00      0,00      0,00
13:05:19        all      2,00      8,01     89,99      0,00      0,00      0,00
```

Вне зависимости от количество воркеров (2, 4, 8, 16) процессор всегда был загружен на уровне ядра на 86-90%, график не прилагаю по только что описанной причине.

Вывод