Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский университет ИТМО»

Факультет Программной инженерии и компьютерной техники

Лабораторная работа №1

по «Операционным системам»

Выполнил: Группа P33312 Хайкин О. И.

Преподаватель:

Пашин А.Д.

Санкт-Петербург, 2023

Текст 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)

Параметры по варианту:

- cpu
  - int128decimal
  - gray
- cache
  - l1cache-line-size
  - l1cache
- io
  - ioprio
  - iomix
- memory
  - madvise
  - prefetch
- network
  - netlink-proc
  - dccp
- pipe
  - pipeherd
  - oom-pipe
- sched
  - sched-period
  - resched

Шаги выполнения

Мониторинг CPU

Для мониторинга CPU будем работать с параметрами –cpu и –cpu-method. Проверим оба выданных по варианту метода и будем изменять число воркеров для каждого, сохраняя информацию о потреблении CPU и числу bogo ops.

Исполняемая команда будет иметь вид

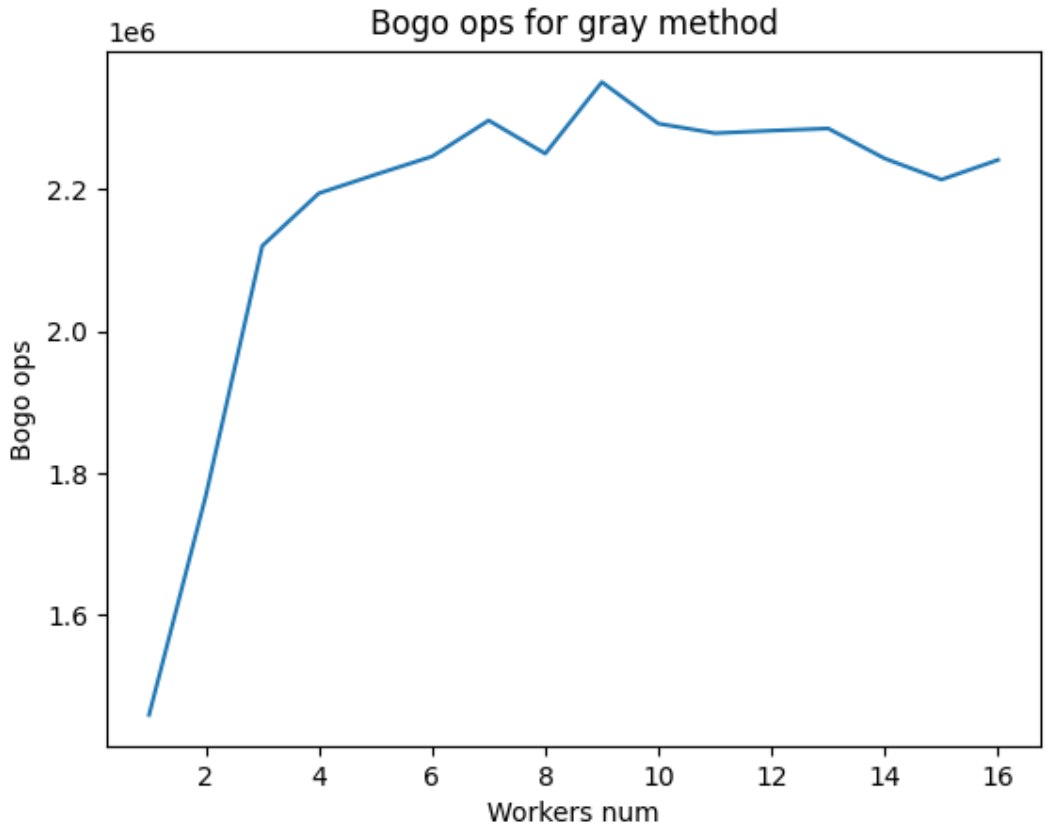
```
$ stress-ng --cpu-method <method> --cpu <worker_num> --timeout 30s --metrics
```

Число bogo ops будем получать от самого stress-ng (с помощью опции –metrics). Потребление CPU будем получать с помощью следующей команды:

```
$ top -b -p<PID-1> -p<PID-2> ... -p<PID-N> -n 1
```

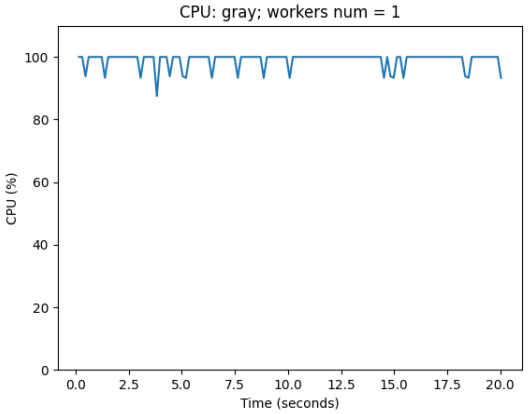
Чтобы не делать всё это ручками, напишем небольшой вспомогательный python-скрипт, который будет запускать stress-ng с разным числов воркеров и мониторить потребление CPU с помощью top, пока он работает. Этим же скриптом соберём статистику по числу bogo ops.

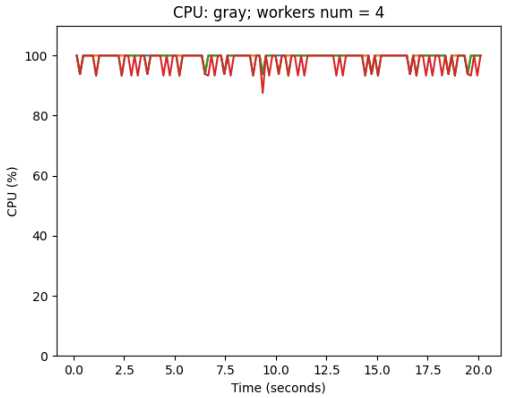
gray

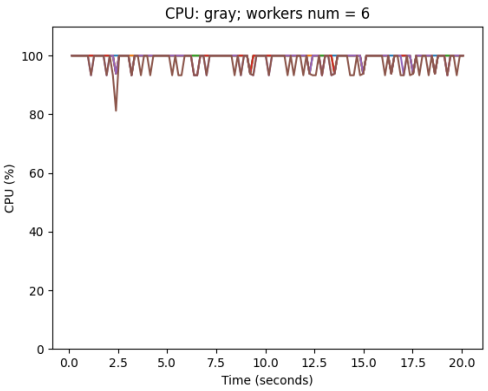


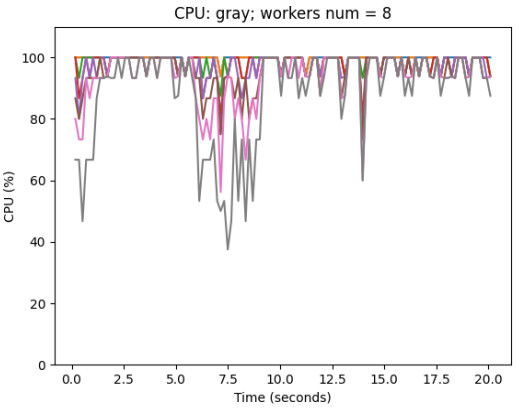
На графике выше видно число bogo ops в зависимости от числа воркеров. Замечаем, что для числа воркеров больше 4-ёх число операций стагнирует.

Рассмотрим также потребление CPU на следующих графиках: (CPU% от времени, на графиках показаны значения для всех воркеров, которые работали одновременно)

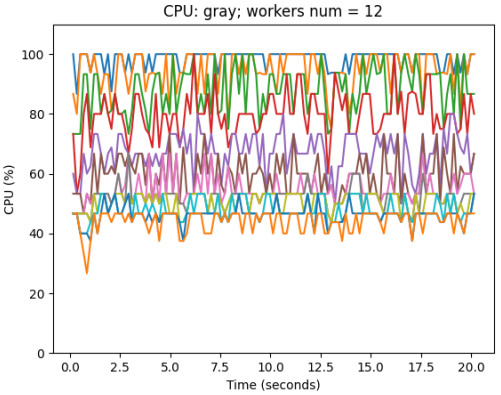


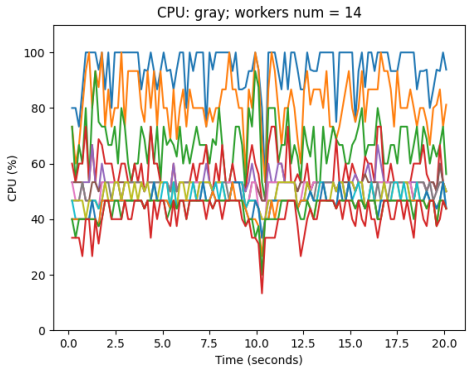


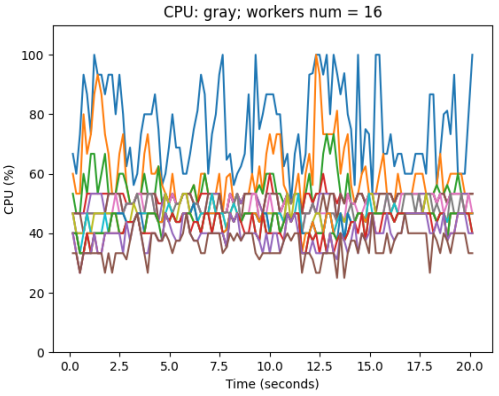




Для числа воркеров от 1-го до 8-ми наблюдается похожая картина - потребление CPU всеми воркерами находится в области 100%, за исключением моментарных провалов. Для 8-ми воркеров появляются более существенные провалы, но в целом тренд оставался таким же. Ситуация меняется, если мы посмотрим на графики для числа воркеров больше 8-ми:





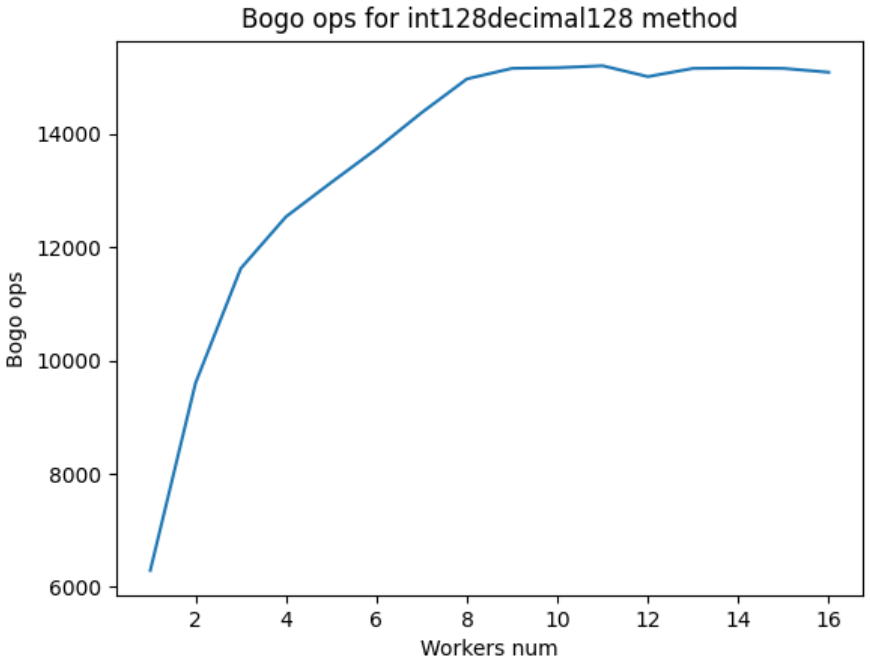


Для числа воркеров больше 8-ми появляются воркеры, потребление CPU которыми находится в области 50-60% - при этом стоит заметить, что при переходе от 8-ми воркеров к 9-воркерам на 50% “опустились” 2 воркера, а не один добавленный.

Для большего числа воркеров всё больше из них “опускаются” на уровни около 50%. К 16 воркерам уже не остаётся ни одного воркера, работающего в зонах 100% потребления CPU.

Прежде чем делать выводы, рассмотрим, наблюдается ли подобная ситуация для другого cpu-метода

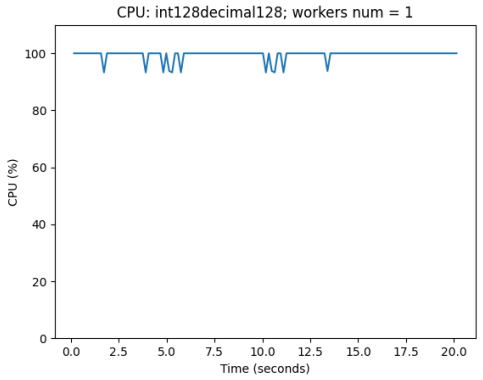
int128decimal128

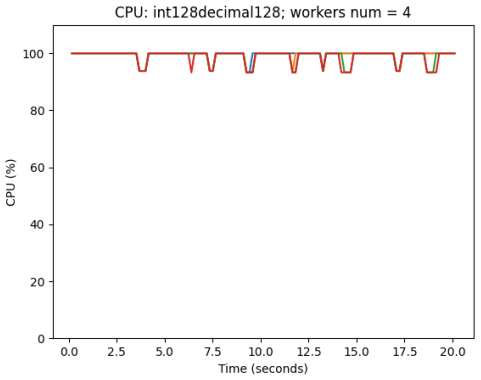


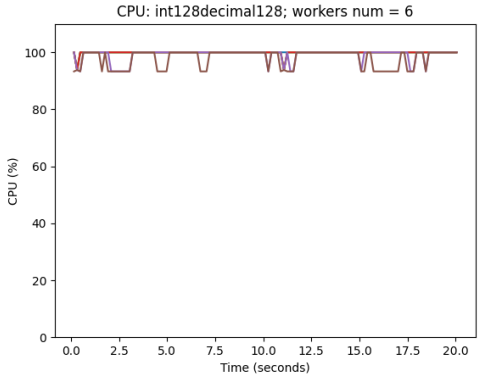
Первое, что сразу становится заметно, это куда более низкие числа bogo ops, в сравнении с первым методом - наблюдается различие на несколько порядков (для gray-метода - 10^6, для int128decimal128 - 10^3)

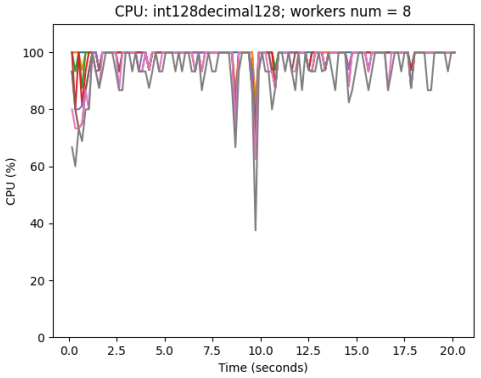
Второе, что становится заметно, это несколько отличающийся тренд - стагнация начинается с 8-ми воркеров, а не 4-ёх, как для gray-метода

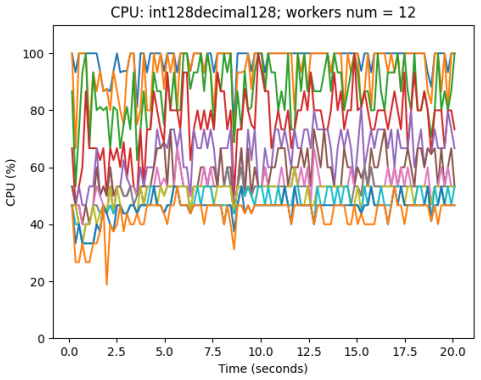
Рассмотрим потребление CPU:

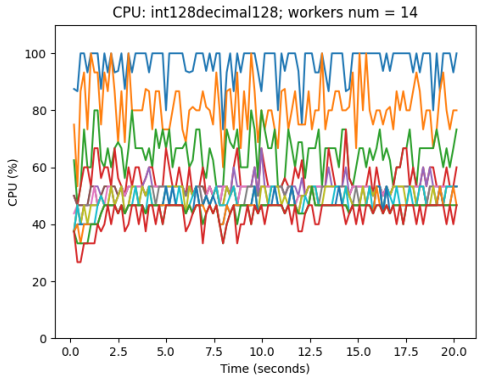


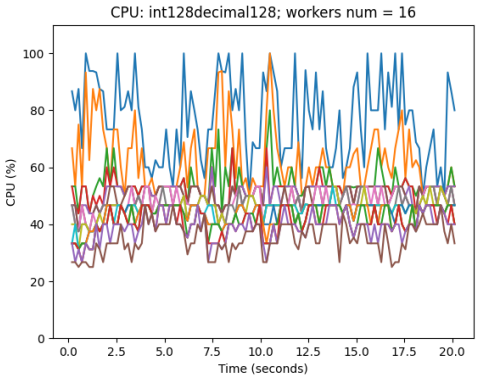












Наблюдаем аналогичную ситуацию - для числа воркеров от 1-го до 8-ми потребление CPU находится в районе 100%, после чего воркеры “проваливаются” на уровни ~50%.

Выводы

Скорее всего, наблюдаемый тренд потребления CPU, вызван особенностями процессора испытуемого компьютера. Рассмотрим их:

```
$ 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:            Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz
    CPU family:          6
    Model:               126
    Thread(s) per core:  2
    Core(s) per socket:  4
    Socket(s):           1
    ...
```

Заметим, что процессор содержит 4 ядра и 2 потока на каждое - что согласуется с 8-ю воркерами, которые работали одновременно на уровне ~100%.

Мониторинг Cache

Для мониторинга кэша будем работать с параметрами --l1cache (число воркеров) и --l1cache-line-size (размер строки l1-кэша)

Исполняемая команда будет иметь вид:

```
$ stress-ng --l1cache <worker_num>  --l1cache-line-size <line_size>
--timeout 30s
```

Воспользуемся perf’ом, чтобы рассмотреть статистику по событиям, связанным с l1-кэшом. Итоговая команда будет иметь вид:

```
$ sudo perf stat -e l1d.replacement,l1d_pend_miss.pending stress-ng
--l1cache <worker_num>  --l1cache-line-size <line_size> --timeout 30s
--metrics
```

Значение l1cache-line-size по умолчанию совпадает с реальным системным - в моём случае 64.

```
$ sudo perf stat -e l1d.replacement,l1d_pend_miss.pending stress-ng
--l1cache 1 --l1cache-line-size 64 --timeout 30s
stress-ng: info:  [717721] setting to a 30 second run per stressor
stress-ng: info:  [717721] dispatching hogs: 1 l1cache
stress-ng: info:  [717722] stress-ng-l1cache: l1cache: size: 48.0K,
sets: 64, ways: 12, line size: 64
stress-ng: info:  [717721] successful run completed in 30.01s

 Performance counter stats for 'stress-ng --l1cache 1
--l1cache-line-size 64 --timeout 30s':

         1 580 726      l1d.replacement
        31 466 159      l1d_pend_miss.pending

      30,014702273 seconds time elapsed

      30,012076000 seconds user
       0,000000000 seconds sys
```

```
$ sudo perf stat -e l1d.replacement,l1d_pend_miss.pending stress-ng
--l1cache 1 --l1cache-line-size 16 --timeout 30s
stress-ng: info:  [717734] setting to a 30 second run per stressor
stress-ng: info:  [717734] dispatching hogs: 1 l1cache
stress-ng: info:  [717735] stress-ng-l1cache: l1cache: size: 48.0K,
sets: 256, ways: 12, line size: 16
stress-ng: info:  [717734] successful run completed in 30.07s

 Performance counter stats for 'stress-ng --l1cache 1
--l1cache-line-size 16 --timeout 30s':

         3 741 682      l1d.replacement
        69 420 114      l1d_pend_miss.pending

      30,083490386 seconds time elapsed

      30,066403000 seconds user
       0,012553000 seconds sys
```

```
$ sudo perf stat -e l1d.replacement,l1d_pend_miss.pending stress-ng
--l1cache 1 --l1cache-line-size 8 --timeout 30s
stress-ng: info:  [717747] setting to a 30 second run per stressor
stress-ng: info:  [717747] dispatching hogs: 1 l1cache
stress-ng: info:  [717748] stress-ng-l1cache: l1cache: size: 48.0K,
sets: 512, ways: 12, line size: 8
stress-ng: info:  [717747] successful run completed in 30.28s
```

```
 Performance counter stats for 'stress-ng --l1cache 1
--l1cache-line-size 8 --timeout 30s':

    40 993 252 863      l1d.replacement
   228 398 162 074      l1d_pend_miss.pending

      30,282691263 seconds time elapsed

      30,276715000 seconds user
       0,003999000 seconds sys
```

```
$ sudo perf stat -e l1d.replacement,l1d_pend_miss.pending stress-ng
--l1cache 1 --l1cache-line-size 96 --timeout 30s
stress-ng: info:  [717891] setting to a 30 second run per stressor
stress-ng: info:  [717891] dispatching hogs: 1 l1cache
stress-ng: info:  [717892] stress-ng-l1cache: l1cache: size: 48.0K,
sets: 42, ways: 12, line size: 96
stress-ng: info:  [717891] successful run completed in 30.04s

 Performance counter stats for 'stress-ng --l1cache 1
--l1cache-line-size 96 --timeout 30s':

          668 364      l1d.replacement
       30 936 745      l1d_pend_miss.pending

      30,045871232 seconds time elapsed

      30,034808000 seconds user
       0,008110000 seconds sys
```

```
$ sudo perf stat -e l1d.replacement,l1d_pend_miss.pending stress-ng
--l1cache 1 --l1cache-line-size 128 --timeout 30s
stress-ng: info:  [717937] setting to a 30 second run per stressor
stress-ng: info:  [717937] dispatching hogs: 1 l1cache
stress-ng: info:  [717938] stress-ng-l1cache: l1cache: size: 48.0K,
sets: 32, ways: 12, line size: 128
stress-ng: info:  [717937] successful run completed in 30.05s

 Performance counter stats for 'stress-ng --l1cache 1
--l1cache-line-size 128 --timeout 30s':

          715 828      l1d.replacement
       24 183 156      l1d_pend_miss.pending
```

```
     30,062010370 seconds time elapsed

     30,049975000 seconds user
      0,008998000 seconds sys
```

Заметим, что для значений размера строки равного 64 или больше, изучаемые значения остаюся примерно равными, но для значений меньше наблюдается увеличение.

Выводы

Число 64 имело “смысл” по аппаратным причинам: это реальная длина строки l1-кэша на моёём устройстве:

```
$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 49152
LEVEL1_DCACHE_ASSOC                12
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  524288
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  6291456
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC
LEVEL4_CACHE_LINESIZE
```

Мониторинг IO

Для мониторинга IO будем работать с параметрами –ioprio и –iomix. Оба эти параметра задают число воркеров для операций, причём в отличии от всех предыдущих конфигураций воркеров, iomix создаёт несколько (19) процессов на одного воркера.

Исполняемая команда будет иметь вид

```
$ stress-ng -<ioprio/iomix> <worker_num> -timeout <timeout>
```

Будем измерять чтение/запись спомощью команды pidstat:

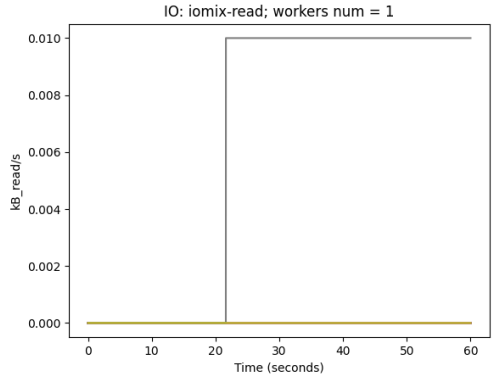
```
$ pidstat -d -p `pgrep stress-ng | sed '1d' | tr '\n' ',' | sed
's/.$//'`
```

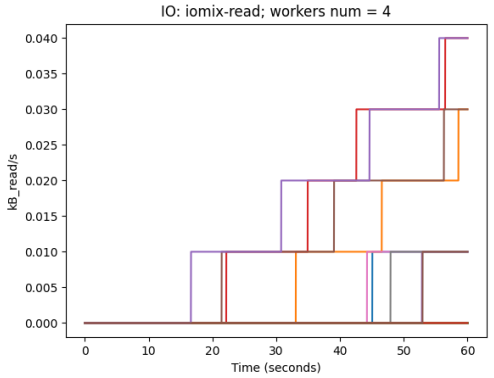
(это вариант команды в одну строчку - будет выполнен pidstat для всех процессов stress-ng, кроме основного)

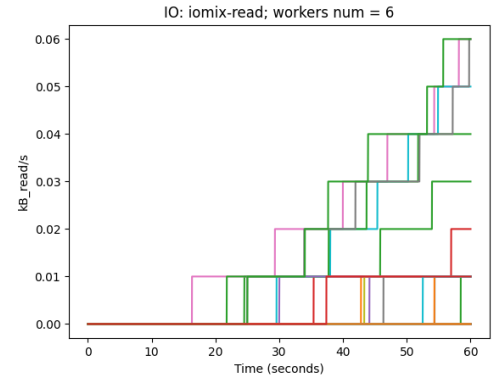
Чтобы посмотреть на результаты, воспользуемся очередным python-скриптом, который запустит stress-ng и периодически будет замерять параметры с помощью pidstat

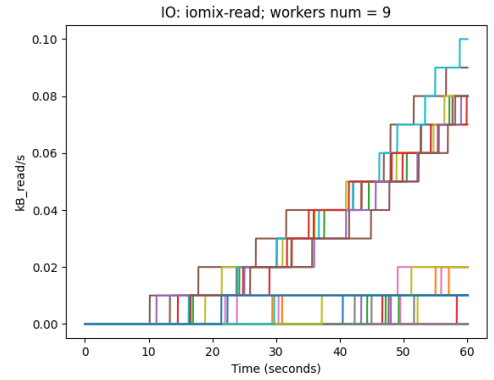
iomix

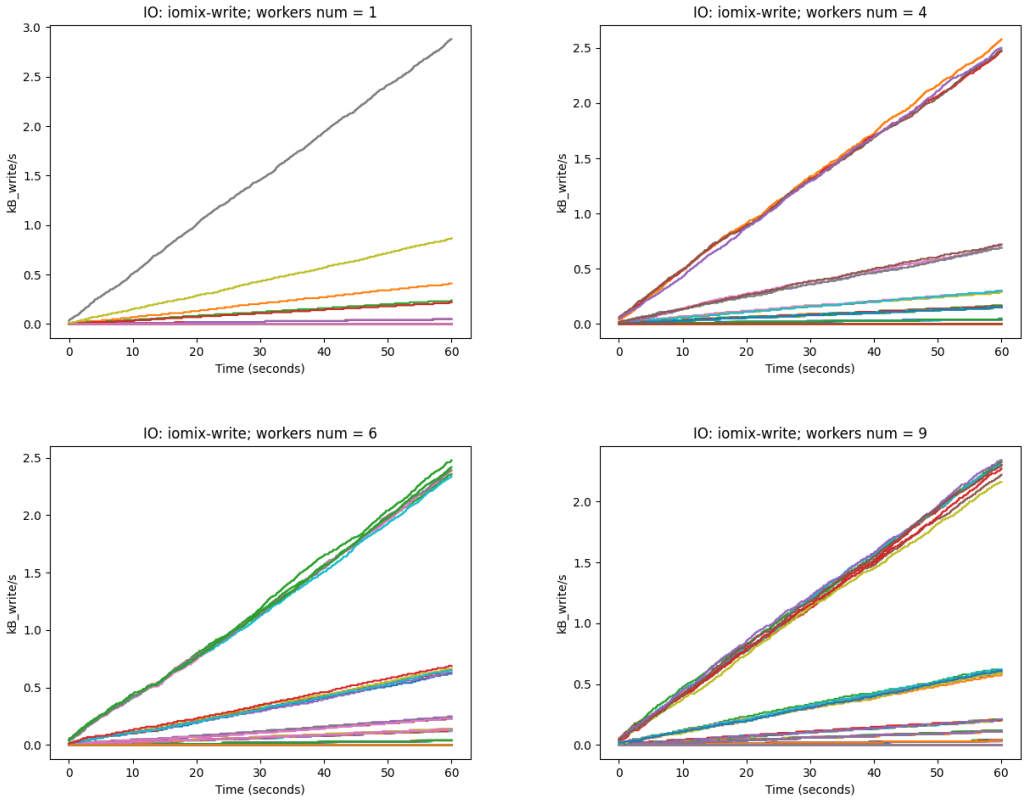
Ниже приведены графики чтения и записи для разного числа воркеров:



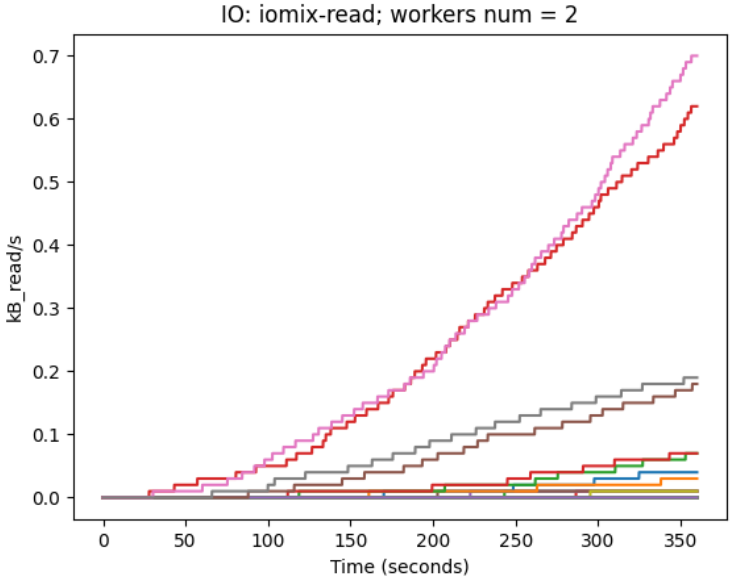


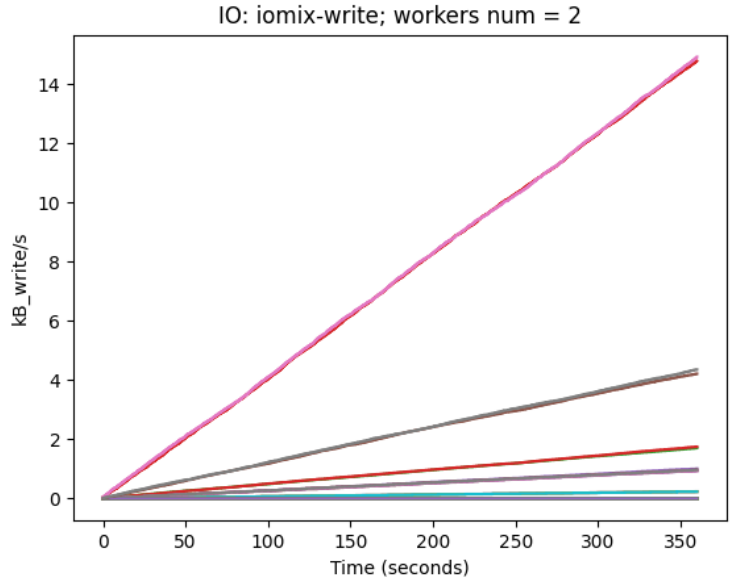




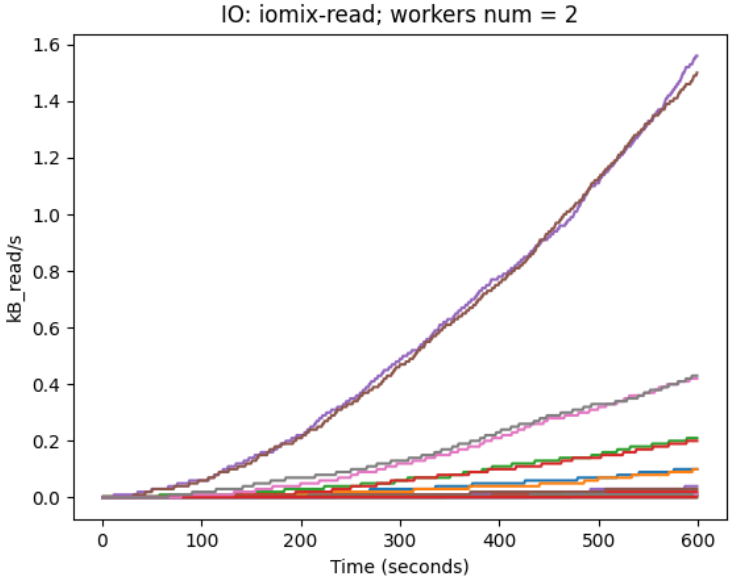


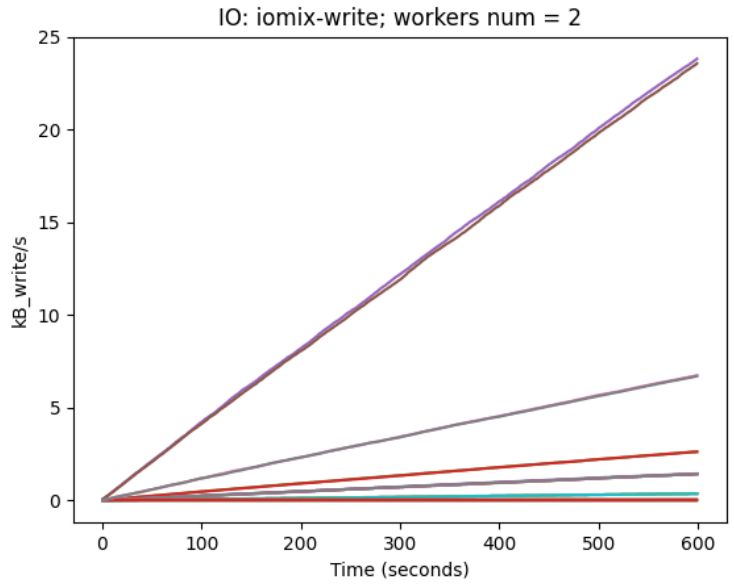
Как видно, тренды не меняются от числа воркеров - с течением времени, скорость чтения/записи растёт - судя по всему, имплементация stress-ng постепенно наращивает скорость. Попробуем продлить время работы программы.





После запуска на 6 минут изменений в тренде не наблюдается.

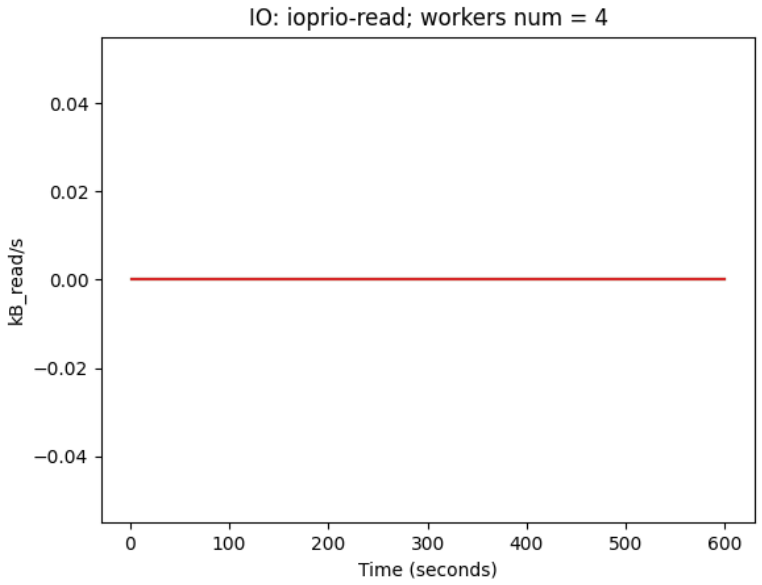


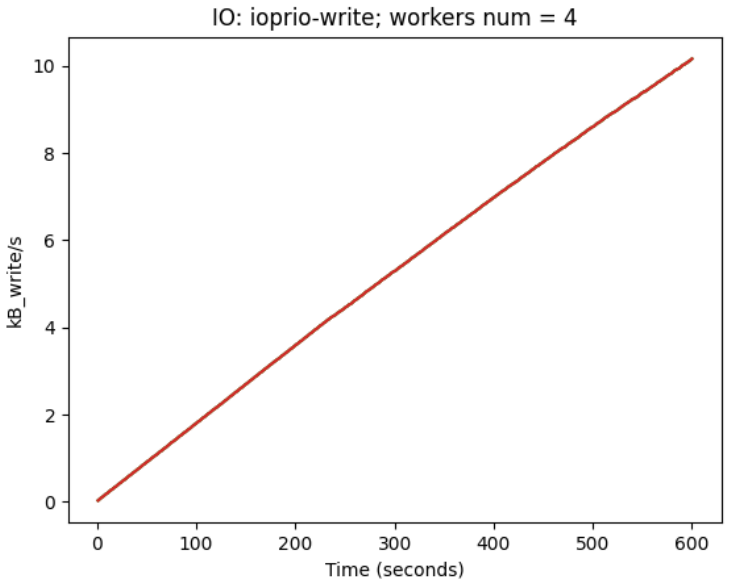


После запуска на 10 минут изменений тоже не видно. Какое-то “предельное” значение скорости тоже не нашлось

ioprio

Рассмотрим аналогичный сценарий для --ioprio





Для ioprio можно гарантировано сказать, что чтения не происходит, а запись одинакова для всех воркеров.

Выводы

Единственный вывод, который здесь удалось получить, это то что io-тесты постепенно увеличивают нагрузку, и верхней границы этой нагрузки за разумное время обнаружить не удалось

Мониторинг Memory

Для мониторинга memory воспользуемся параметрами --madvise и --prefetch

Посмотрим на потребление воркерами CPU, используя скрипт из мониторинга CPU

madvise

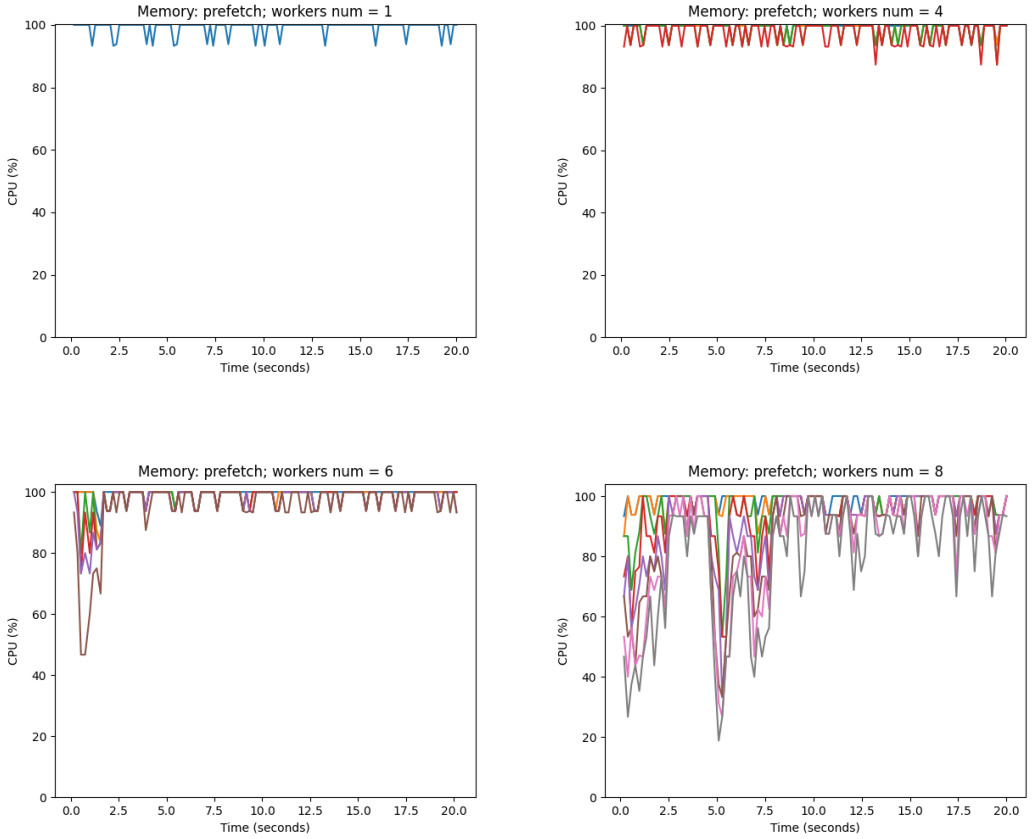
Заметим, что процессы madvise стабильно занимают больше 100% CPU. Это означает, что эти процессы работают на нескольких ядрах процессора. Разумеется, не всякий процесс может работать на нескольких ядрах. Проверив исходный код, можно увидеть следующий блок:

```
{
	pthread_t pthreads[NUM_PTHREADS];
	int rets[NUM_PTHREADS];
	size_t i;

	ctxt.is_thread = true;

	for (i = 0; i < NUM_PTHREADS; i++) {
		rets[i] = pthread_create(&pthreads[i], NULL,
					stress_madvise_pages, (void *)&ctxt);
	}
	for (i = 0; i < NUM_PTHREADS; i++) {
		if (rets[i] == 0)
```

```
                (void)pthread_join(pthreads[i], NULL);
        }
}
```

T.e. воркеры madvise создают несколько (8) потоков, что и объясняет, почему эти процессы могут занимать больше 100% CPU.

prefetch



Для prefetch картина похожа на cpu-воркеры - здесь замечаний нет.

Мониторинг Network

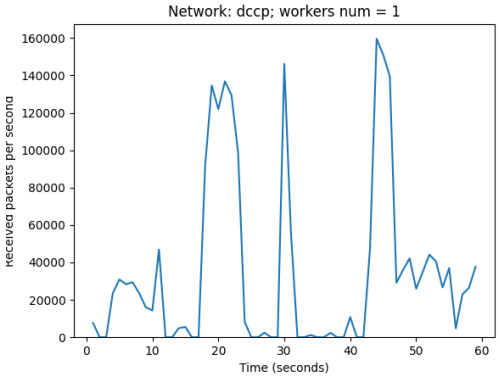
Для мониторинга network воспользуемся параметрами --netlink-proc и --dccp.

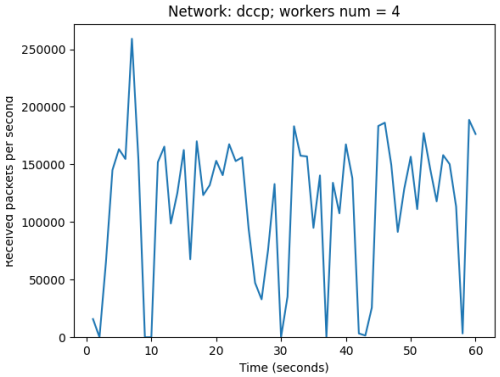
Чтобы узнать число переданных и полученных пакетов воспользуемся утилитой ip.

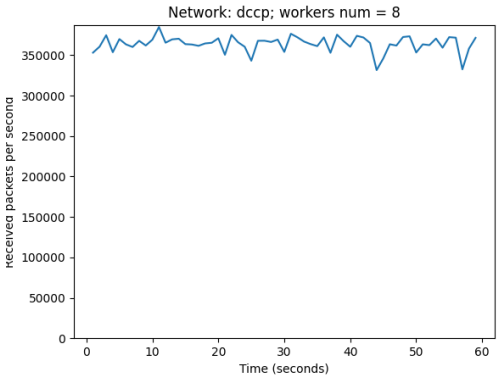
dccp

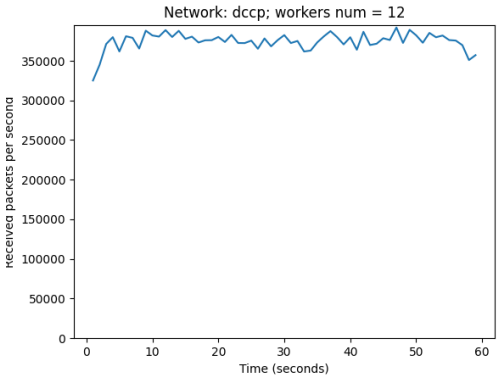
Рассмотрим статистику по пакетам для различного числа воркеров:

Полученные пакеты

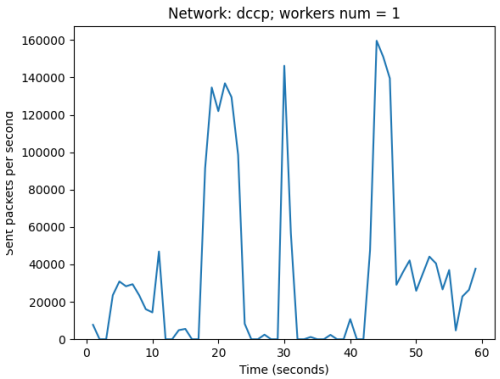


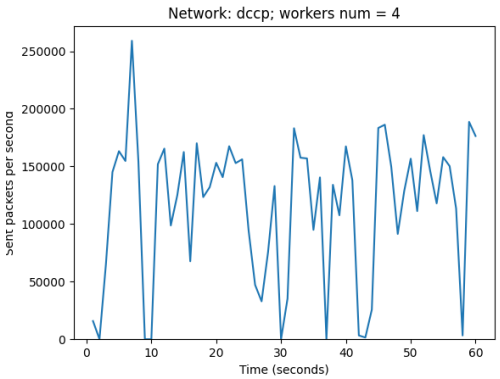


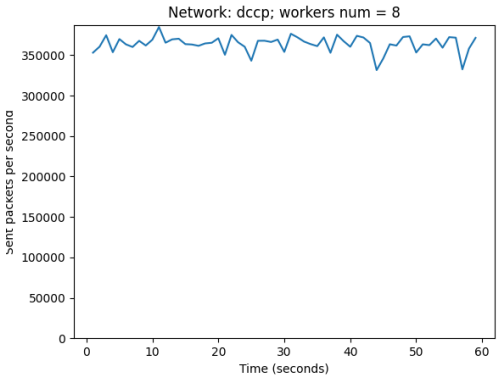


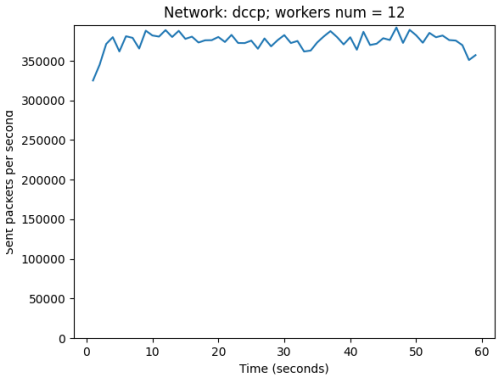


Переданные пакеты









Заметим, что начиная с 8 воркеров число полученных/переданных пакетов в секунду стагнирует в районе 350000 пакетов/сек.

netlink-proc

netlink-proc сам по себе не совсем связан с network'ом - этот тест создаёт несколько потоков и мониторит вызовы fork/exec/exit, считая каждый вызов bogo op'ом. Для мониторинга тест подключается к proc netlink interface. Проверим, что это и вправду происходит:

```
$ stress-ng: info:  [257422] defaulting to a 86400 second (1 day, 0.00
secs) run per stressor
stress-ng: info:  [257422] dispatching hogs: 1 netlink-proc

$ pgrep stress-ng
257422
257423

$ cat /proc/net/netlink | grep 257423
sk               Eth Pid        Groups   Rmem     Wmem     Dump  Locks
Drops    Inode
0000000000000000 11  257423     00000001 39168    0        0     2
87282    145147
```

Видим, что в списке netlink-сокетов есть сокет, связанный с stress-ng процессом. Также видим высокое значение Rmem и нулевое значение Wmem - т.е. процесс занимается только чтением через сокет.

Рассмотрев исходный код теста, находим следующий блок:

```
struct nlmsghdr *nlmsghdr;
ssize_t len;
char __attribute__ ((aligned(NLMSG_ALIGNTO)))buf[4096];
```

```
if ((len = recv(sock, buf, sizeof(buf), 0)) == 0)
      return 0;

if (len == -1) {
      switch (errno) {
      case EINTR:
      case ENOBUFS:
            return 0;
      default:
            return -1;
      }
}
```

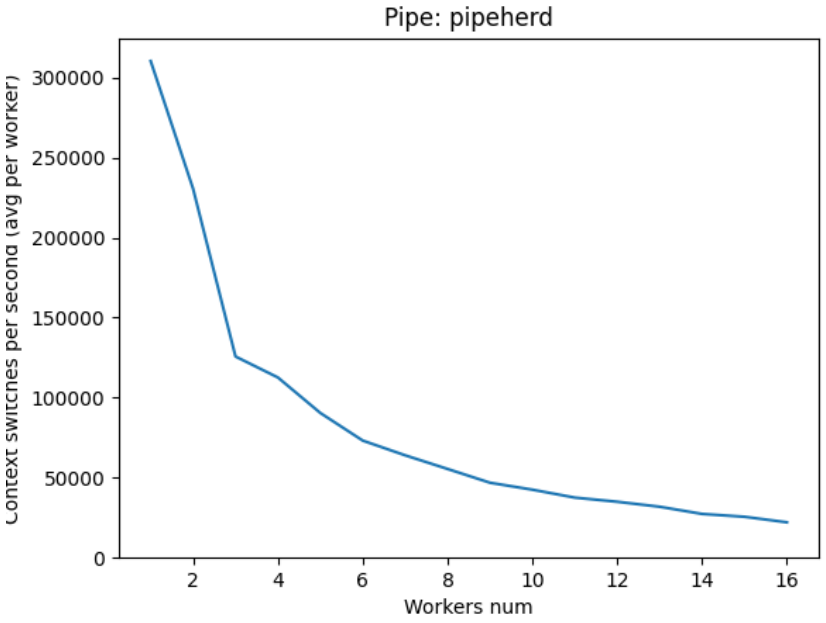
Это единственный блок, в котором созданный сокет используется - что подтверждает нашу теорию о чтении.

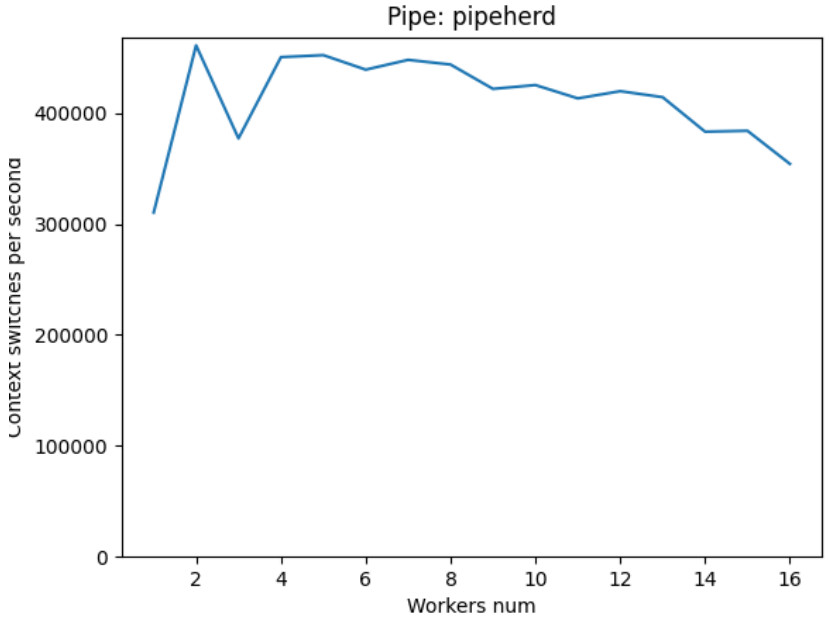
Мониторинг Pipe

Для мониторинга pipe воспользуемся параметрами --pipeherd и --oom-pipe.

pipeherd

Параметр pipeherd выведет число context-switch'ей в секунду/на один bogo op, посмотрим на это значение для разного числа воркеров (среднее для каждого воркера и суммарное для всех воркеров):





Как видно из графиков, общее число context-switch'ей остаётся примерно постоянным, вне зависимости от числа воркеров.

oom-pipe

Этот параметр создаёт наибольшее возможно число pipe'ов и начинает уменьшать и увеличивать их, тем самым производя большое число аллокаций, что заставит ядро начать убивать процессы по причине Out Of Memory. Посмотрим и убедимся, что это происходит:

Логи ядра до запусска команды (есть 3 записи об убийстве с прошлых тестов):

```
$ grep -i 'killed process' /var/log/kern.log
Oct 16 11:42:48 ... kernel: [ 6110.689055] Out of memory: Killed process
14044 (stress-ng) ...
Oct 16 11:42:53 ... kernel: [ 6116.169329] Out of memory: Killed process
14043 (stress-ng) ...
Oct 16 11:42:54 ... kernel: [ 6117.670236] Out of memory: Killed process
14042 (stress-ng) ...
```

Запустим стресс-нг с параметром oom-pipe:

```
$ stress-ng --pathological --oom-pipe 1 --timeout 1m
```

Проверяем логи через короткий промежуток времени:

```
$ grep -i 'killed process' /var/log/kern.log
Oct 16 11:42:48 ... kernel: [ 6110.689055] Out of memory: Killed process
14044 (stress-ng) ...
```

```
Oct 16 11:42:53 ... kernel: [ 6116.169329] Out of memory: Killed process
14043 (stress-ng) ...
Oct 16 11:42:54 ... kernel: [ 6117.670236] Out of memory: Killed process
14042 (stress-ng) ...
```

Видим, что ничего не изменилось. Попробуем увеличить число воркеров, чтобы получить более агрессивную нагрузку:

```
$ stress-ng --pathological --oom-pipe 16 --timeout 1m &
```

Отмечаем, что компьютеру становится тяжело (заметны сильные провисания), и проверяем логи по окончанию действия программы:

```
$ grep -i 'killed process' /var/log/kern.log
Oct 16 11:42:48 ... kernel: [ 6110.689055] Out of memory: Killed process
14044 (stress-ng) ...
Oct 16 11:42:53 ... kernel: [ 6116.169329] Out of memory: Killed process
14043 (stress-ng) ...
Oct 16 11:42:54 ... kernel: [ 6117.670236] Out of memory: Killed process
14042 (stress-ng) ...

Oct 16 17:54:08 ... kernel: [28391.230274] Out of memory: Killed process
88487 (stress-ng) ...
Oct 16 17:54:08 ... kernel: [28391.238201] Out of memory: Killed process
88465 (stress-ng) ...
Oct 16 17:54:08 ... kernel: [28391.247059] Out of memory: Killed process
88489 (stress-ng) ...
Oct 16 17:54:08 ... kernel: [28391.252183] Out of memory: Killed process
88484 (stress-ng) ...
Oct 16 17:54:08 ... kernel: [28391.304034] Out of memory: Killed process
88491 (stress-ng) ...
Oct 16 17:54:08 ... kernel: [28391.333912] Out of memory: Killed process
88490 (stress-ng) ...
Oct 16 17:54:19 ... kernel: [28400.928814] Out of memory: Killed process
88478 (stress-ng) ...
Oct 16 17:54:20 ... kernel: [28402.950322] Out of memory: Killed process
88472 (stress-ng) ...
Oct 16 17:54:20 ... kernel: [28403.070453] Out of memory: Killed process
88470 (stress-ng) ...
Oct 16 17:54:20 ... kernel: [28403.138778] Out of memory: Killed process
88505 (stress-ng) ...
Oct 16 17:54:28 ... kernel: [28410.774622] Out of memory: Killed process
88486 (stress-ng) ...
Oct 16 17:54:28 ... kernel: [28411.641887] Out of memory: Killed process
88485 (stress-ng) ...
Oct 16 17:54:29 ... kernel: [28411.944481] Out of memory: Killed process
```

```
88504 (stress-ng) ...
Oct 16 17:54:30 ... kernel: [28412.916183] Out of memory: Killed process
88503 (stress-ng) ...
Oct 16 17:54:40 ... kernel: [28422.383575] Out of memory: Killed process
88512 (stress-ng) ...
Oct 16 17:54:41 ... kernel: [28424.187902] Out of memory: Killed process
88488 (stress-ng) ...
Oct 16 17:54:42 ... kernel: [28424.658232] Out of memory: Killed process
88476 (stress-ng) ...
Oct 16 17:54:47 ... kernel: [28430.426807] Out of memory: Killed process
88473 (stress-ng) ...
Oct 16 17:54:54 ... kernel: [28437.586339] Out of memory: Killed process
88523 (stress-ng) ...
Oct 16 17:54:59 ... kernel: [28441.767693] Out of memory: Killed process
88469 (stress-ng) ...
```

В действительности видим, что произошло много OOM-убийств

Мониторинг Sched

resched

Воспользуемся perf'ом чтобы собрать статистику по schedule до запуска stress-ng:

```
$ sudo perf sched record sleep 2
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 3,133 MB perf.data (7699 samples) ]
```

```
$ sudo perf sched script
            perf 545457 [000] 30718.258518: sched:sched_stat_runtime:
comm=perf pid=545457 runtime=26717 [ns] vruntime=49075519352831 [ns]
            perf 545457 [000] 30718.258521:       sched:sched_waking:
comm=migration/0 pid=15 prio=0 target_cpu=000
            perf 545457 [000] 30718.258522: sched:sched_stat_runtime:
comm=perf pid=545457 runtime=4630 [ns] vruntime=49075519357461 [ns]
            perf 545457 [000] 30718.258523:       sched:sched_switch:
prev_comm=perf prev_pid=545457 prev_prio=120 prev_state=R+ ==>
next_comm=migration/0 next_pid=15 next_prio=0
     migration/0    15 [000] 30718.258524: sched:sched_migrate_task:
comm=perf pid=545457 prio=120 orig_cpu=0 dest_cpu=1
```

```
     migration/0    15 [000] 30718.258529:       sched:sched_switch:
prev_comm=migration/0 prev_pid=15 prev_prio=0 prev_state=S ==>
next_comm=swapper/0 next_pid=0 next_prio=120
            perf 545457 [001] 30718.258559: sched:sched_stat_runtime:
comm=perf pid=545457 runtime=33571 [ns] vruntime=48534856980401 [ns]
            perf 545457 [001] 30718.258561:       sched:sched_waking:
comm=migration/1 pid=21 prio=0 target_cpu=001
            perf 545457 [001] 30718.258562: sched:sched_stat_runtime:
comm=perf pid=545457 runtime=3680 [ns] vruntime=48534856984081 [ns]
            perf 545457 [001] 30718.258563:       sched:sched_switch:
prev_comm=perf prev_pid=545457 prev_prio=120 prev_state=R+ ==>
next_comm=migration/1 next_pid=21 next_prio=0
     migration/1    21 [001] 30718.258565: sched:sched_migrate_task:
comm=perf pid=545457 prio=120 orig_cpu=1 dest_cpu=2
     migration/1    21 [001] 30718.258570:       sched:sched_switch:
prev_comm=migration/1 prev_pid=21 prev_prio=0 prev_state=S ==>
next_comm=swapper/1 next_pid=0 next_prio=120
         swapper     0 [000] 30718.258600:       sched:sched_switch:
prev_comm=swapper/0 prev_pid=0 prev_prio=120 prev_state=R ==>
next_comm=pool-tracker-st next_pid=545464 next_prio=120
 pool-tracker-st 545464 [000] 30718.258620: sched:sched_stat_runtime:
comm=pool-tracker-st pid=545464 runtime=23472 [ns]
vruntime=49075519380933 [ns]
 pool-tracker-st 545464 [000] 30718.258622:       sched:sched_switch:
prev_comm=pool-tracker-st prev_pid=545464 prev_prio=120 prev_state=D ==>
next_comm=swapper/0 next_pid=0 next_prio=120
...
```

Видим, что было собрано ~3 MB информации, и большая её часть относится к самому perf. Попробуем собрать эти же показатели с запущенным stress-ng:

```
$ stress-ng --resched 1 &
stress-ng: info:  [545609] defaulting to a 86400 second (1 day, 0.00
secs) run per stressor
stress-ng: info:  [545609] dispatching hogs: 1 resched

$ sudo perf sched record sleep 2
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 1007,783 MB perf.data (8941402
samples) ]
```

Сразу замечаем, что было собрано ~1000 MB информаци. Посмотрим на неё:

```
$ sudo perf sched script
            perf 560236 [000] 30948.803823: sched:sched_stat_runtime:
comm=perf pid=560236 runtime=35158 [ns] vruntime=49794566708612 [ns]
```

```
            perf 560236 [000] 30948.803827:       sched:sched_waking:
comm=migration/0 pid=15 prio=0 target_cpu=000
            perf 560236 [000] 30948.803828: sched:sched_stat_runtime:
comm=perf pid=560236 runtime=6072 [ns] vruntime=49794566714684 [ns]
            perf 560236 [000] 30948.803830:       sched:sched_switch:
prev_comm=perf prev_pid=560236 prev_prio=120 prev_state=R+ ==>
next_comm=migration/0 next_pid=15 next_prio=0
     migration/0    15 [000] 30948.803833: sched:sched_migrate_task:
comm=perf pid=560236 prio=120 orig_cpu=0 dest_cpu=1
     migration/0    15 [000] 30948.803836: sched:sched_stat_runtime:
comm=stress-ng pid=562385 runtime=1297 [ns] vruntime=49319616235889 [ns]
     migration/0    15 [000] 30948.803838:       sched:sched_switch:
prev_comm=migration/0 prev_pid=15 prev_prio=0 prev_state=S ==>
next_comm=stress-ng next_pid=562419 next_prio=120
       stress-ng 562419 [000] 30948.803841: sched:sched_stat_runtime:
comm=stress-ng pid=562419 runtime=4236 [ns] vruntime=49794577125861 [ns]
       stress-ng 562419 [000] 30948.803842:       sched:sched_switch:
prev_comm=stress-ng prev_pid=562419 prev_prio=120 prev_state=R ==>
next_comm=stress-ng next_pid=562399 next_prio=120
       stress-ng 562399 [000] 30948.803845: sched:sched_stat_runtime:
comm=stress-ng pid=562399 runtime=3559 [ns] vruntime=49794576998778 [ns]
       stress-ng 562399 [000] 30948.803846:       sched:sched_switch:
prev_comm=stress-ng prev_pid=562399 prev_prio=120 prev_state=R ==>
next_comm=stress-ng next_pid=562164 next_prio=127
       stress-ng 562164 [000] 30948.803848: sched:sched_stat_runtime:
comm=stress-ng pid=562164 runtime=3292 [ns] vruntime=49794577603253 [ns]
       stress-ng 562164 [000] 30948.803849:       sched:sched_switch:
prev_comm=stress-ng prev_pid=562164 prev_prio=127 prev_state=R ==>
next_comm=stress-ng next_pid=562399 next_prio=120
       stress-ng 562399 [000] 30948.803850: sched:sched_stat_runtime:
comm=stress-ng pid=562399 runtime=2144 [ns] vruntime=49794577730596 [ns]
       ...
```

Видим, что и вправду появились записи из процессов stress-ng.

Вывод

В ходе выполнения данной лаобраторной работы я познакомился с различными инструментами мониторинга в Linux-системах.