

НИУ ИТМО
ФПИиКТ
Операционные системы

Лабораторная работа №5
Анализ производительности и поведения программ

Передрий Михаил Сергеевич
Р33102

Санкт-Петербург
2024

Задание

Знакомство с системными инструментами анализа производительности и поведения программ. В данной лабораторной работе Вам будет предложено произвести нагружочное тестирование Вашей операционной системы при помощи инструмента stress-ng.

В качестве тестируемых подсистем использовать: cpi, 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).

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.

Вариант

cpi: [int16, int64];
cache: [l1cache-sets, cache-fence];
io: [ioprio, ioport];
memory: [zlib-mem-level, shm];
network: [netdev, sockdiag];
pipe: [pipeherd, pipe-ops];
sched: [sched-deadline, sched-prio].

Выполнение

Все необходимые артефакты (тестовые скрипты, результаты выполнения скриптов, питоновский ноутбук для визуализации, этот отчет) можно найти в репозитории: https://github.com/misuy/itmo_os_lab5.

Система

System Details	
Hardware Information	
Software Information	
Model	Firmware Version
Lenovo Yoga Slim 7 Pro 14ACH5	GZCN20WW
Memory	OS Name
16,0 GiB	Manjaro Linux
Processor	OS Build
AMD Ryzen™ 5 5600H with	rolling
Radeon™ Graphics × 12	OS Type
Graphics	64-bit
AMD Radeon™ Graphics	GNOME Version
Disk Capacity	45.3
Unknown	Windowing System
	X11
	Kernel Version
	Linux 6.1.71-1-MANJARO

```
╭─▶ ↵ ~/Documents/itmo/circuit_design ➤ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:         48 bits physical, 48 bits virtual
Byte Order:            Little Endian
CPU(s):                12
On-line CPU(s) list:  0-11
Vendor ID:             AuthenticAMD
Model name:            AMD Ryzen 5 5600H with Radeon Graphics
CPU family:            25
Model:                 80
Thread(s) per core:   2
Core(s) per socket:   6
Socket(s):             1
Stepping:              0
Frequency boost:      enabled
CPU(s) scaling MHz:  66%
CPU max MHz:          4279,6870
CPU min MHz:          1200,0000
BogoMIPS:              6590,04
```

```
Virtualization features:
Virtualization:        AMD-V
Caches (sum of all):
L1d:                  192 KiB (6 instances)
L1i:                  192 KiB (6 instances)
L2:                   3 MiB (6 instances)
L3:                   16 MiB (1 instance)
NUMA:
NUMA node(s):          1
NUMA node0 CPU(s):     0-11
```

CPU

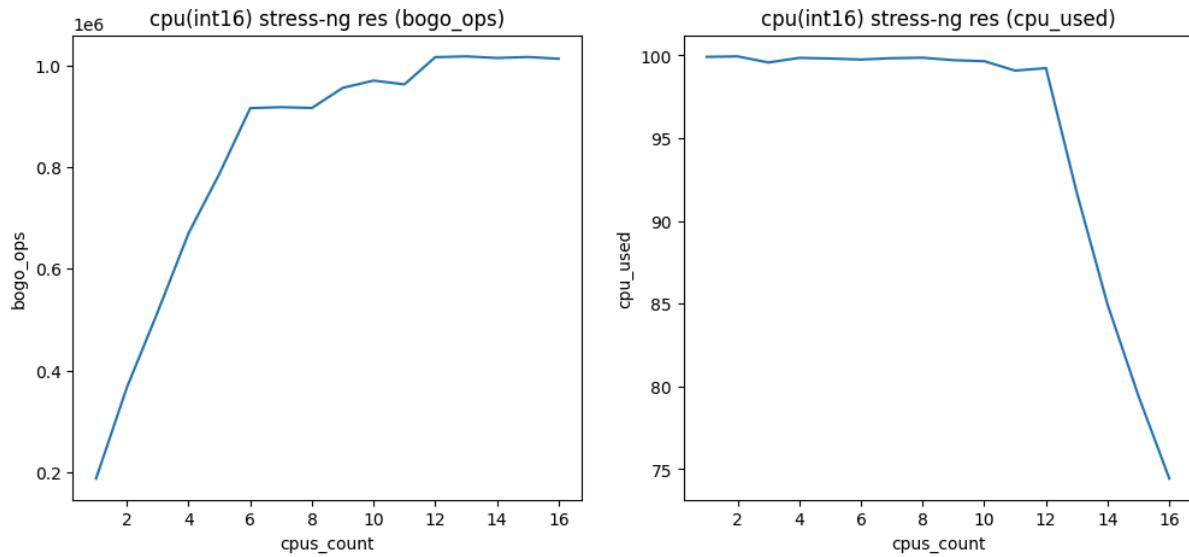
int16 (1000 iterations of a mix of 16 bit integer operations)

Исполняется набор из различных операций над 16-битными целыми числами.

Тестовый скрипт: /cpu/int16

Попробуем запустить бенчмарк с различным количеством воркеров.

```
sudo stress-ng --cpu $i --cpu-method int16 --metrics --timeout  
$timeout
```



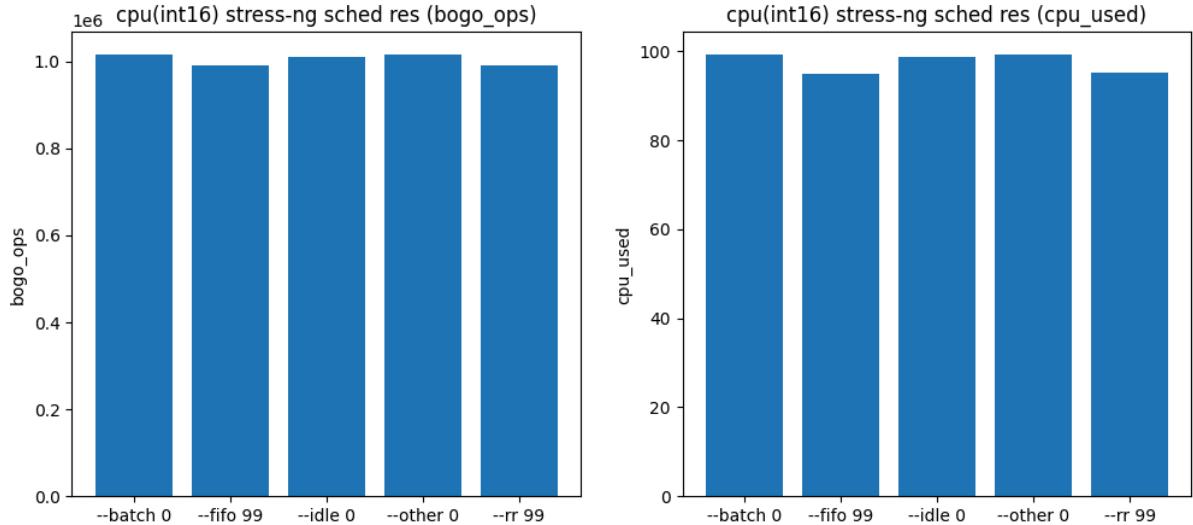
По графику *bogo_ops* видно, что после 12 воркеров начинается плато, что является следствием падения процента процессорного времени, достающегося каждому воркеру (видно по графику *cpu_used*). Это связано с ограничением системы на 12 параллельных потоков.

Аналогичная картина будет наблюдаться почти во всех последующих тестах.

Пик – 1017314 bogo ops.

Зафиксировав оптимальное количество воркеров (12), попробуем дополнительно улучшить результат, используя различные стратегии планирования.

```
policy_list="--batch 0" "--fifo 99" "--idle 0" "--other 0"  
"--rr 99")  
sudo chrt ${policy_list[$policy_idx]} stress-ng --cpu 12  
--cpu-method int16 --metrics --timeout $timeout
```



Получили примерно равные результаты, но в алгоритмах fifi и round-robin процессам почему-то доставалось меньше процессорного времени, что привело к небольшому снижению показателей.

Не получили значительного прироста.

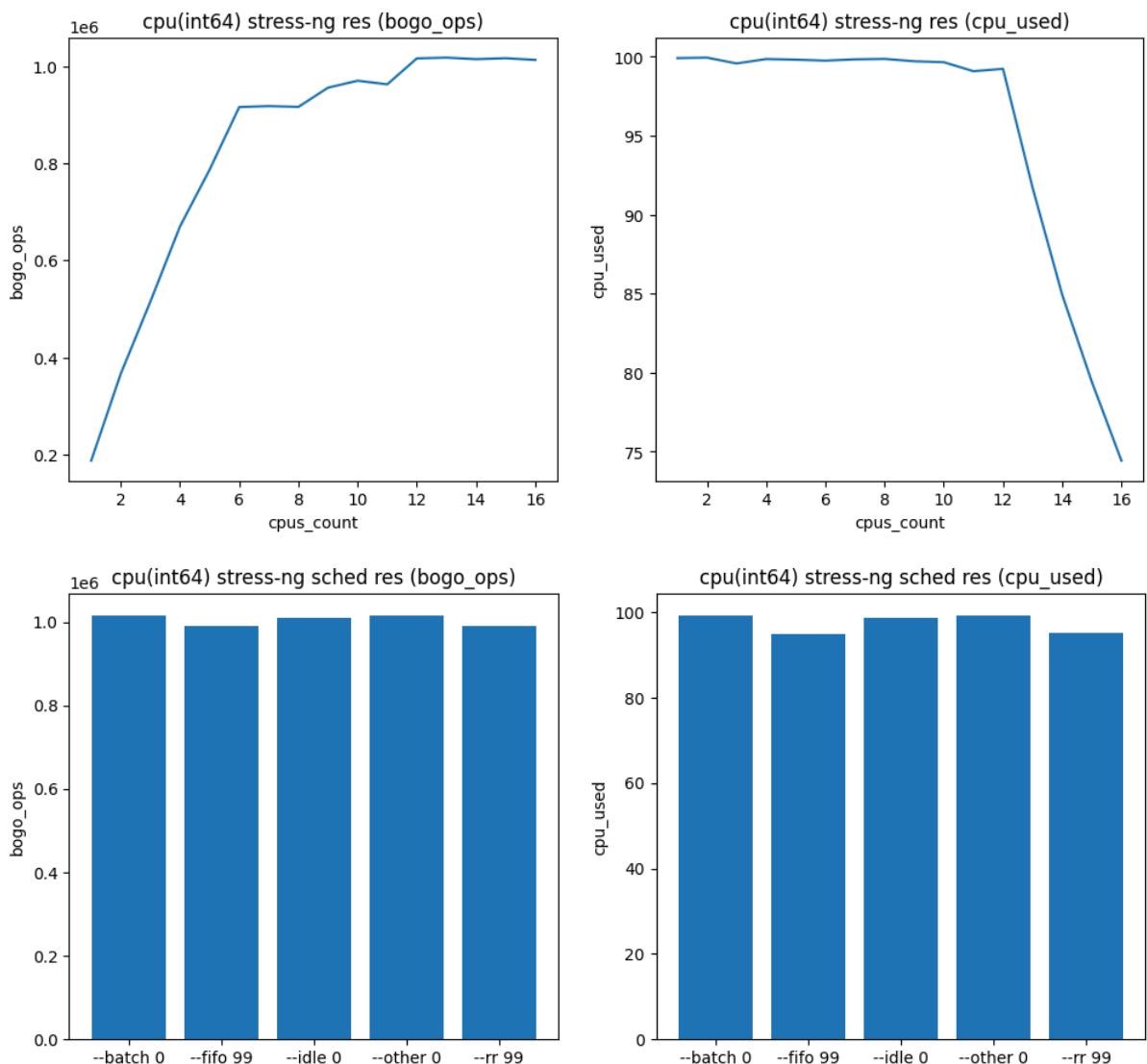
int64 (1000 iterations of a mix of 64 bit integer operations)

Исполняется набор из различных операций над 64-битными целыми числами.

Тестовый скрипт: /cpu/int64

Тест аналогичен предыдущему, поэтому и действовать будем так же.

Результаты:



Пиковое значение – 1074991 bogo ops.

Cache

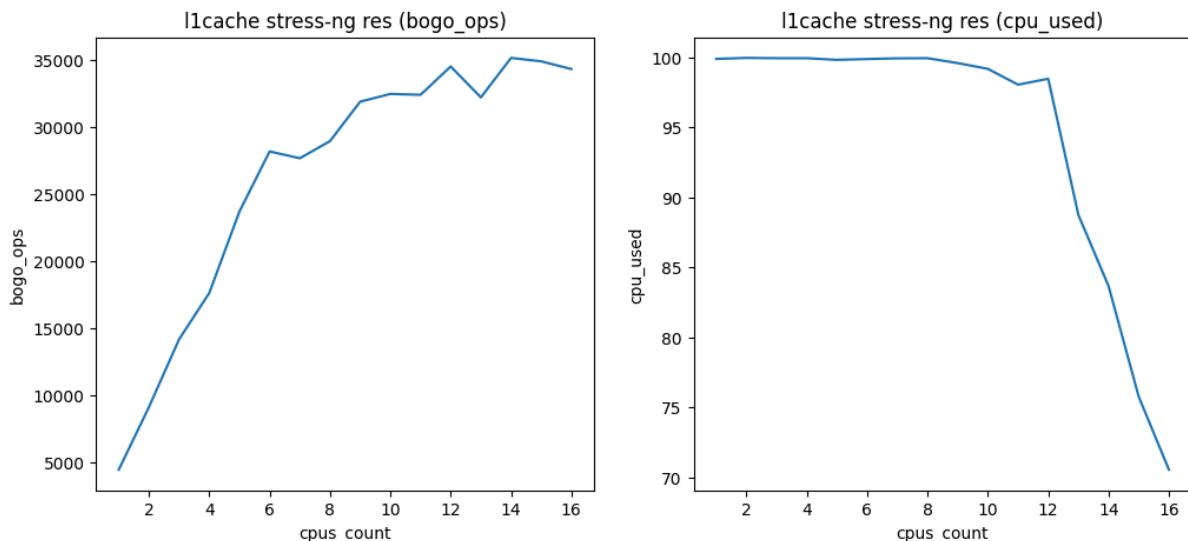
l1cache-sets (specify level 1 cache sets)

Устанавливает количество множеств строк в множественно-ассоциативном кеше.

Тестовый скрипт: `/cache/l1cache-sets`

Для начала выберем оптимальное количество воркеров.

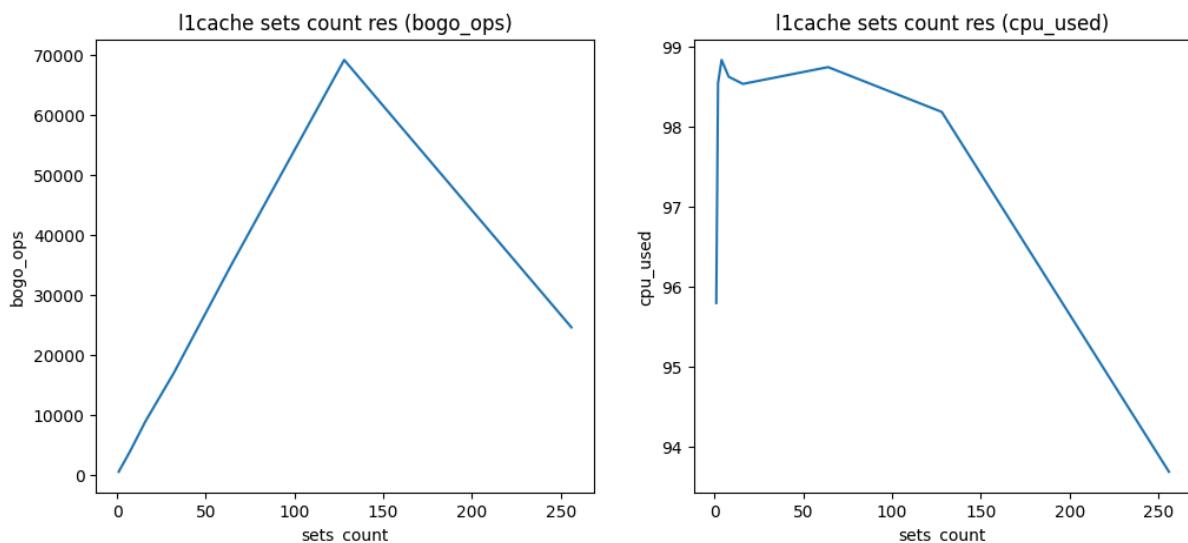
```
sudo stress-ng --l1cache $i --metrics --timeout $timeout
```



Видно, что 12 по-прежнему хороший выбор.

Теперь будем варьировать непосредственно l1cache-sets. Для снижения времени тестирования итерируемся по степеням двойки от 1 до 512.

```
sudo stress-ng --l1cache 12 --l1cache-sets $i --metrics  
--timeout $timeout
```



Лучшие показатели достигнуты при `sets_count ~ 128`.

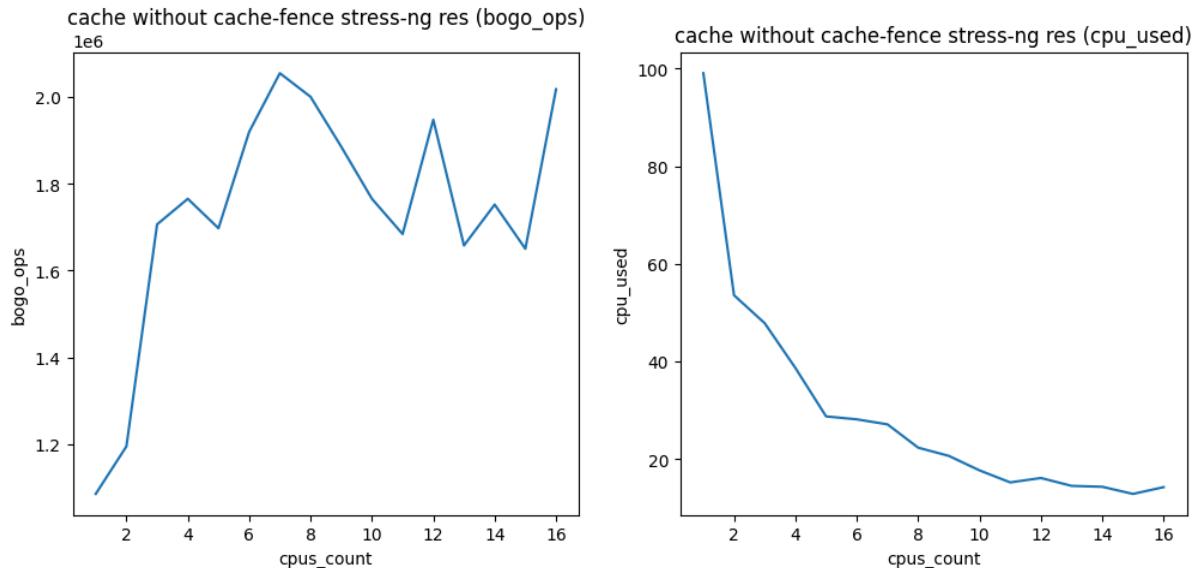
cache-fence (serialize stores)

Использовать барьеры памяти при тестировании кэша.

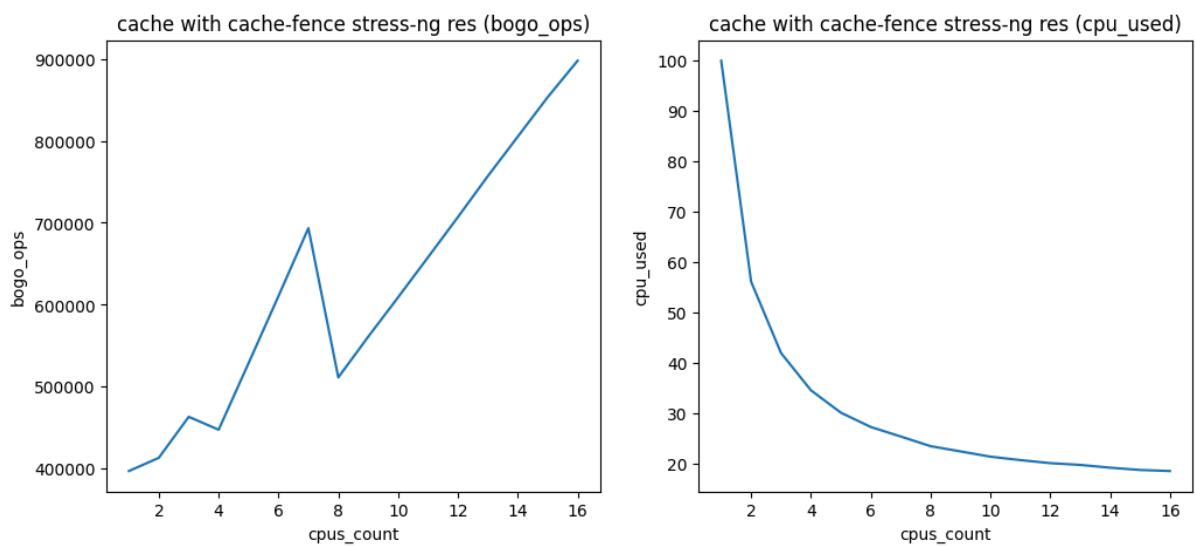
Тестовый скрипт: /cache/cache-fence

Запустим два набора тестов: с включенной опцией и без.

Без cache-fence:



С cache-fence:



С опцией графики стали более “красивыми” (bogo ops похож на линейный, а cpu used почти гипербола). Также видно, что значения bogo ops значительно снизились (более, чем в два раза), что, вероятно, связано с ограничением спекулятивного выполнения.

IO

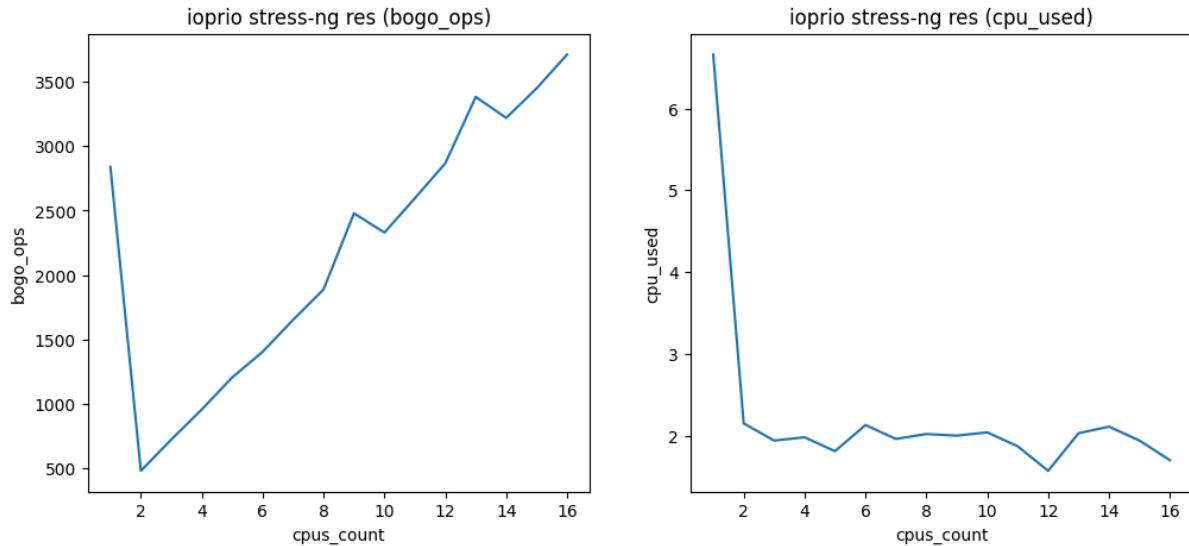
ioprio (start N workers exercising set/get iopriority)

Запуск воркеров, осуществляющих системные вызовы ioprio_get и ioprio_set (изменение приоритетов ввода-вывода).

Тестовый скрипт: /io/ioprio

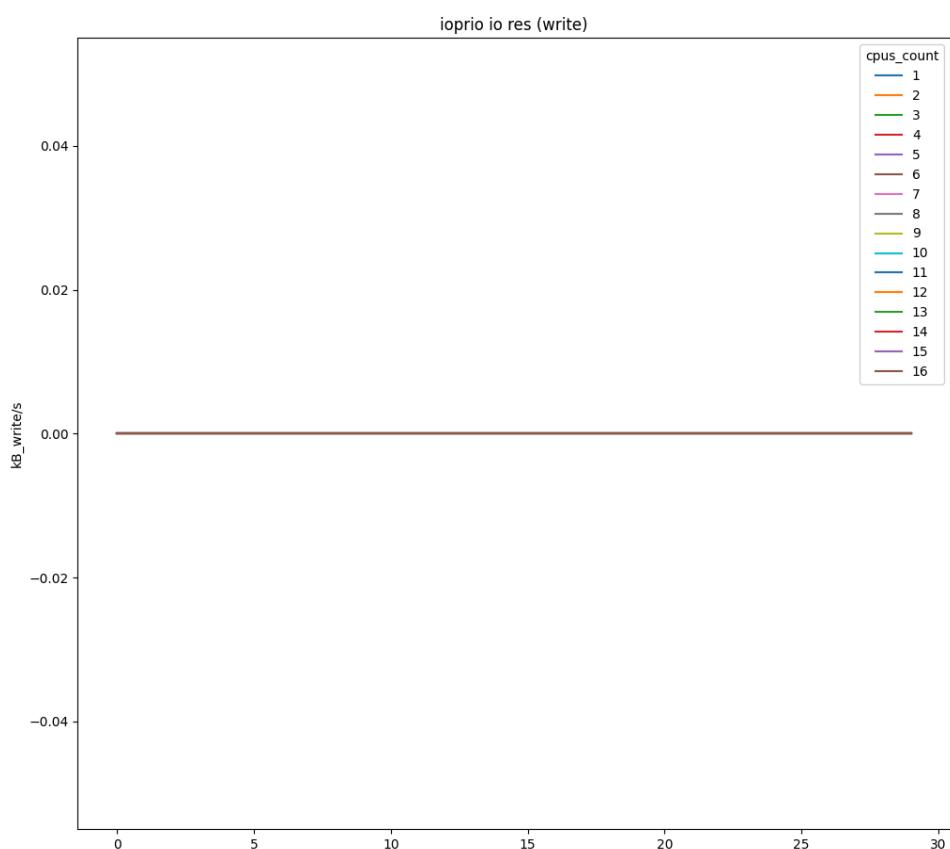
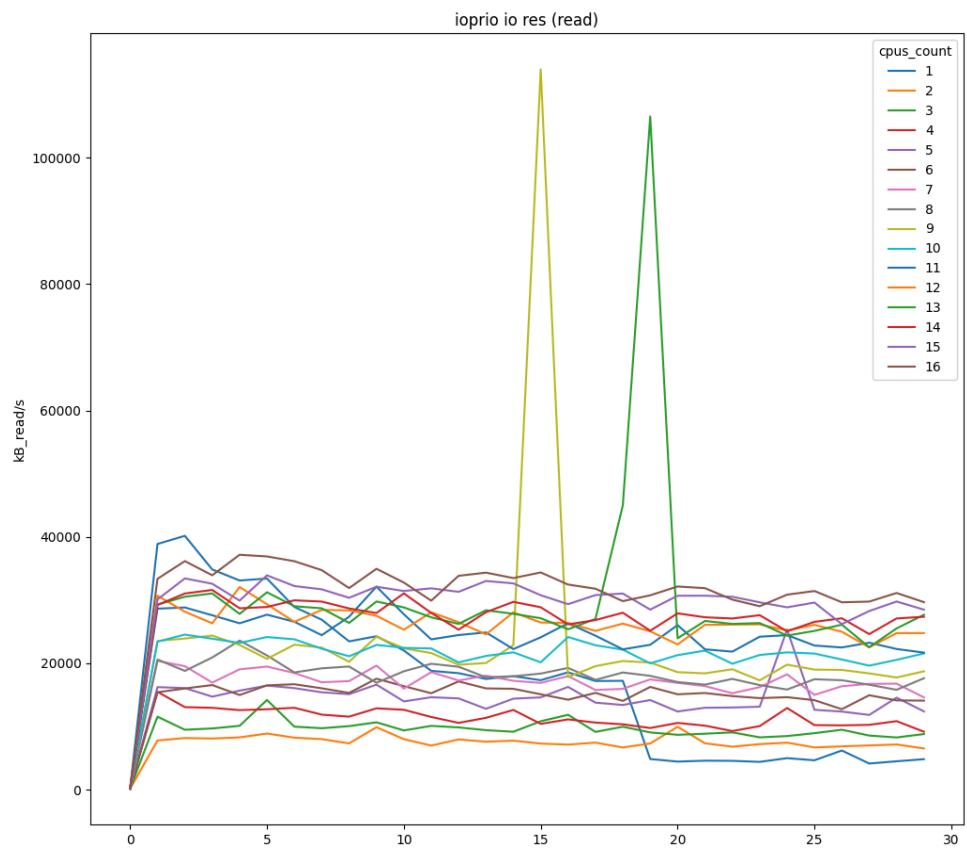
Количество воркеров:

```
sudo stress-ng --ioprio $i --metrics --timeout $timeout
```

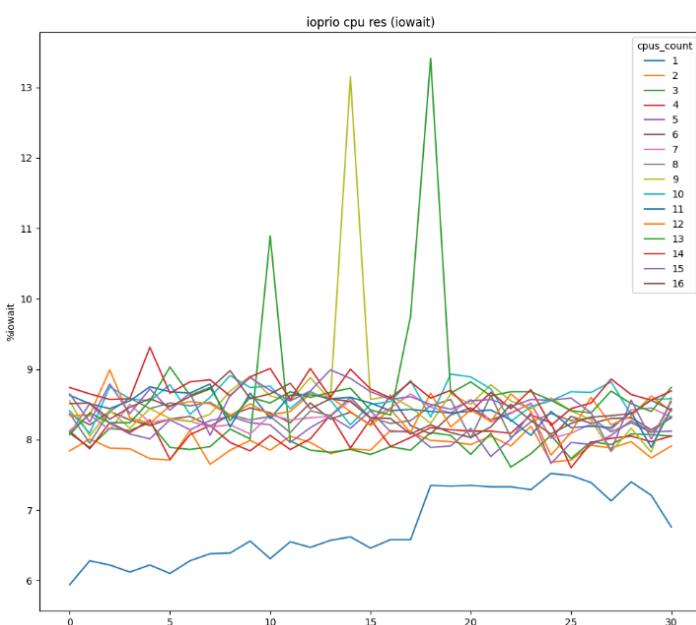
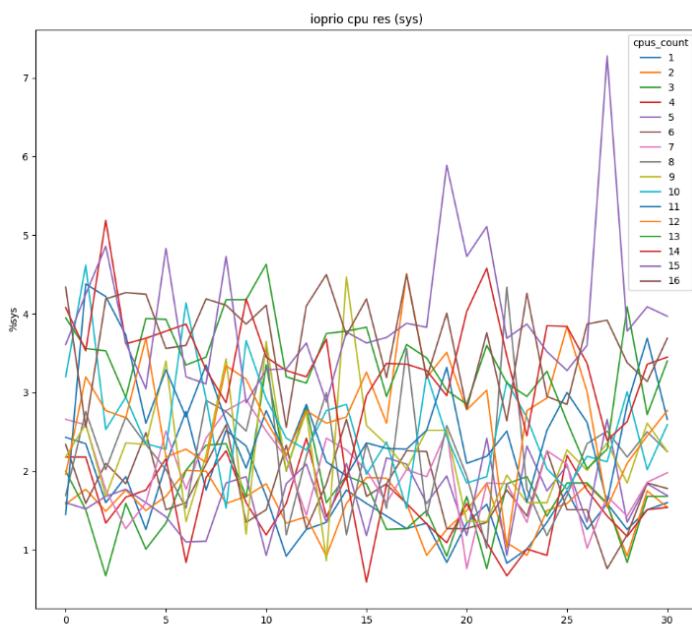
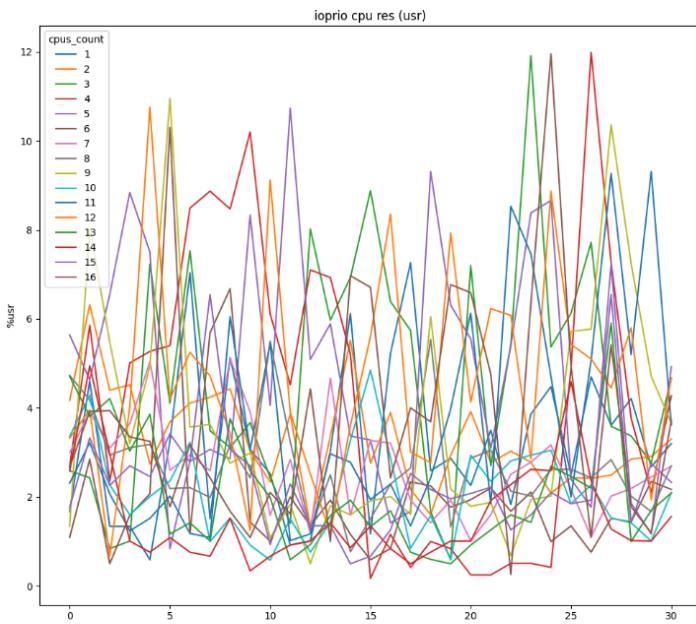


Дополнительно измеряются показатели загрузки процессора (mpstat) и подсистемы ввода вывода (iostat).

Ввод вывод:



Процессор:



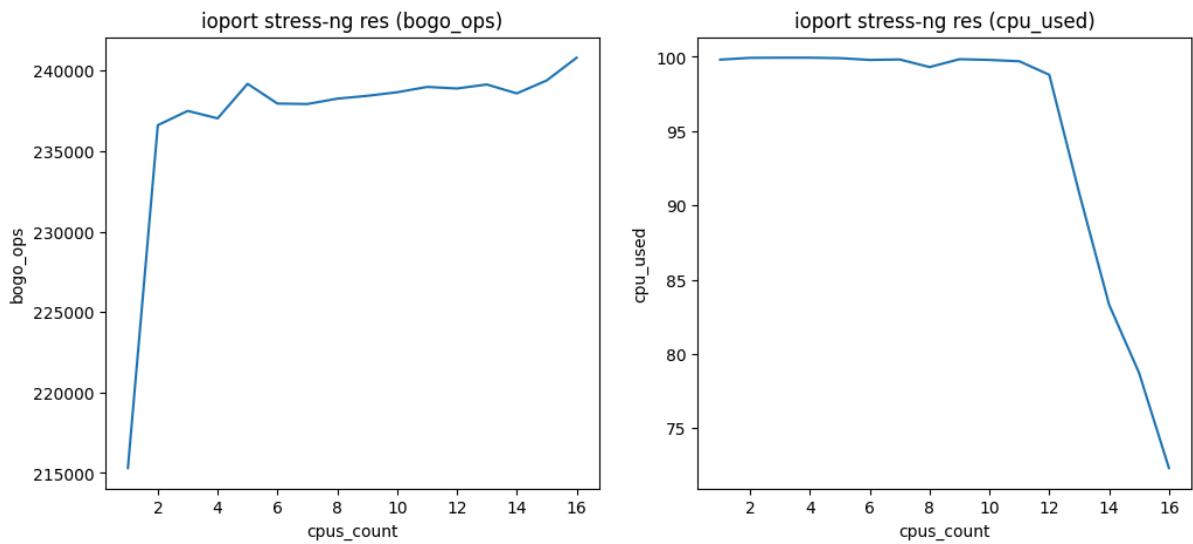
Видно, что процессор не сильно загружен и тратит некоторое время, ожидая выполнение операций ввода-вывода. Кажется, что именно с этим связан линейный рост функции `bogo_ops(cpus_count)`, что позволяет создавать больше воркеров, чем в предыдущих тестах.

Довольно активно происходят чтения.

ioport (start N workers exercising port I/O)

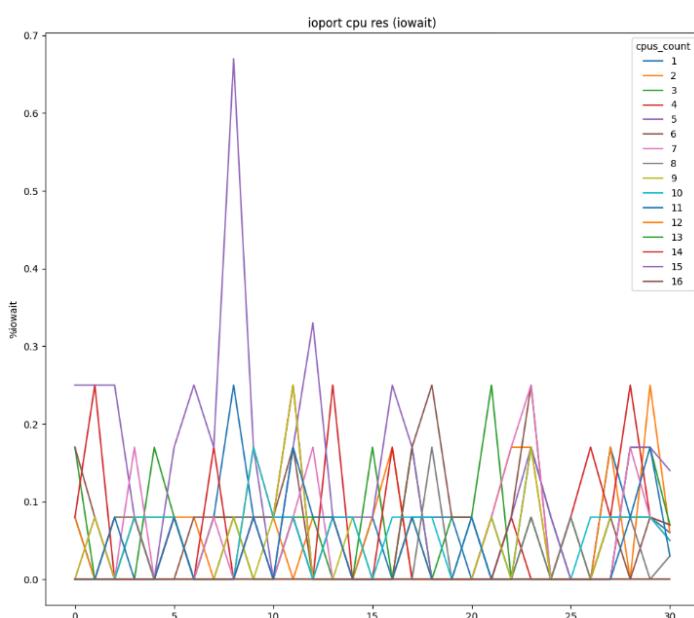
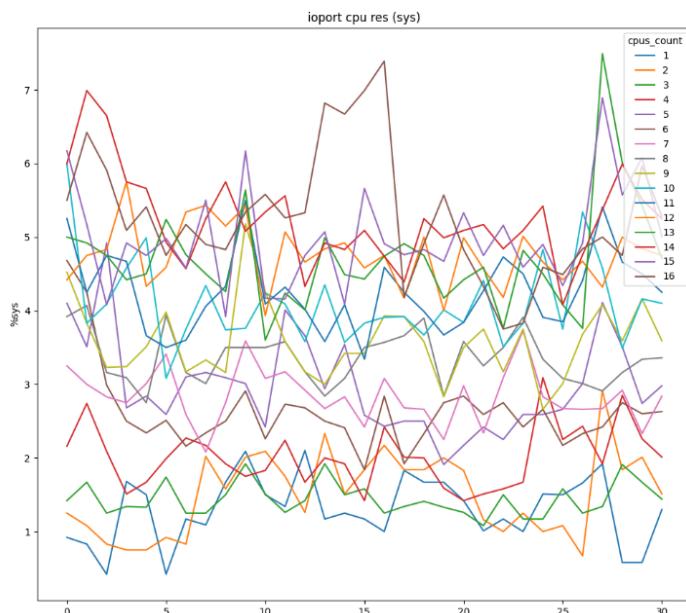
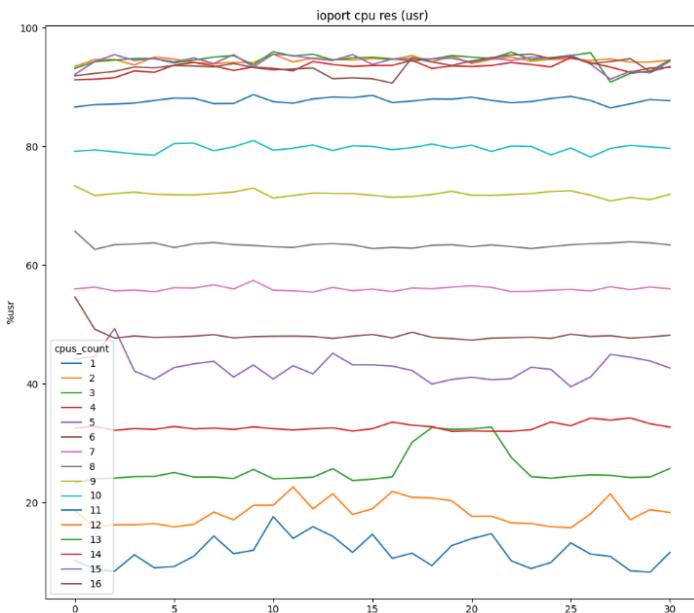
Запуск воркеров, осуществляющих ввод-вывод на порт.

Тестовый скрипт: `/io/ioport`

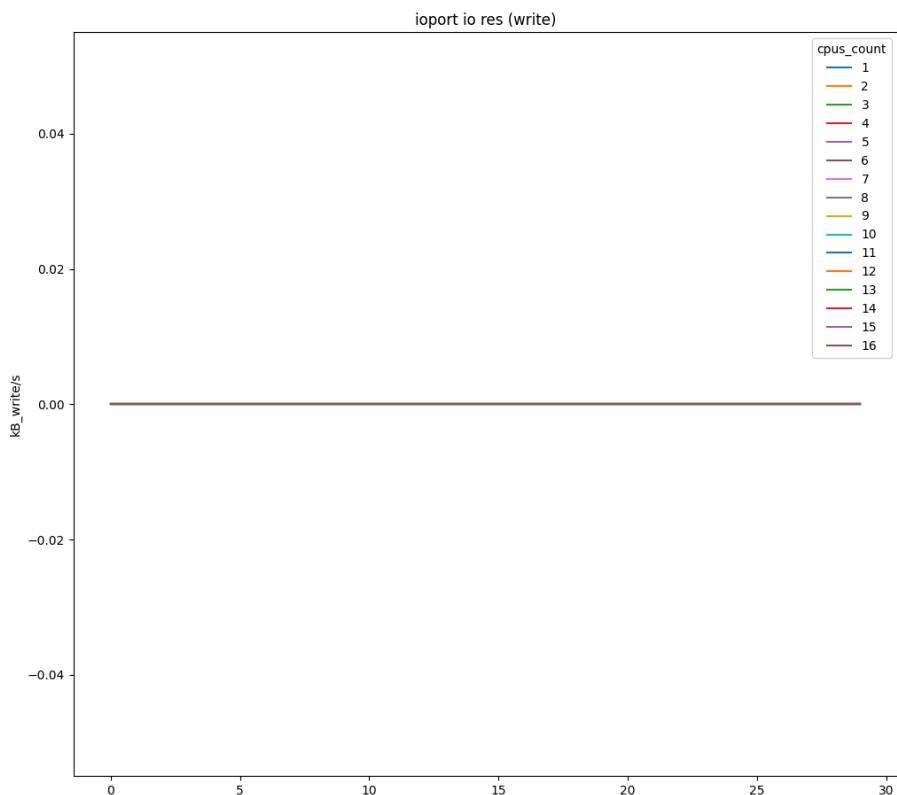
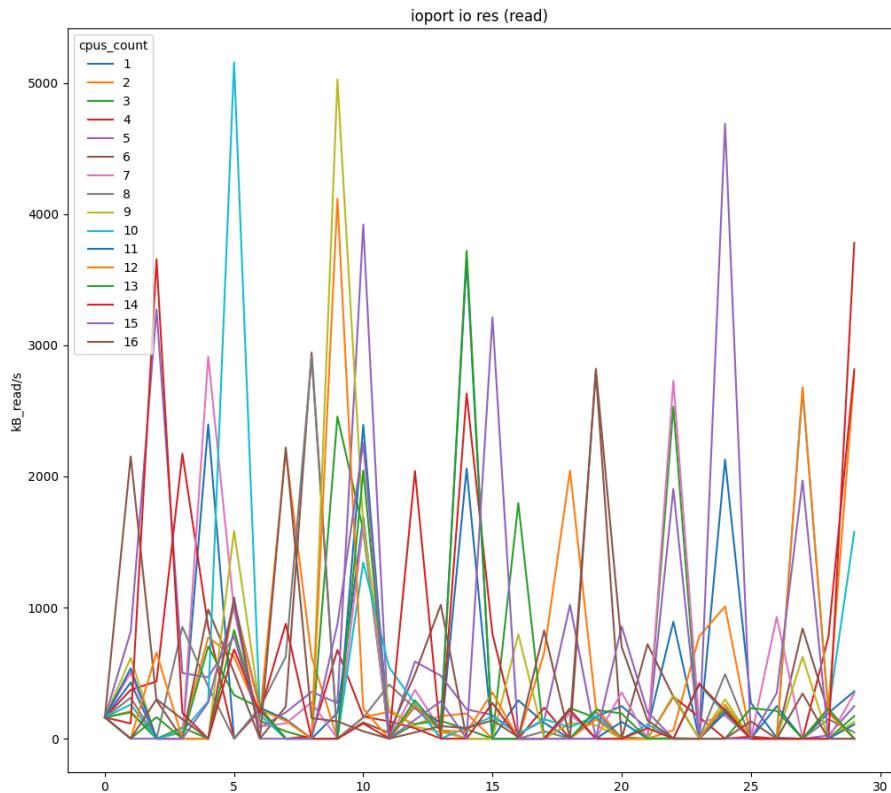


Видим плато, значит, вероятно, воркеры нагружают процессор.

Проверим:



Действительно, ясно виден рост загрузки процессора при увеличении числа воркеров. При этом на ожидание ввода-вывода время практически не расходуется.



Почему-то данный тест слабо нагружает подсистему ввода-вывода (сравнивая с предыдущим), что странно, так как, судя по описанию, должны происходить операции ввода-вывода (может быть их просто очень мало).

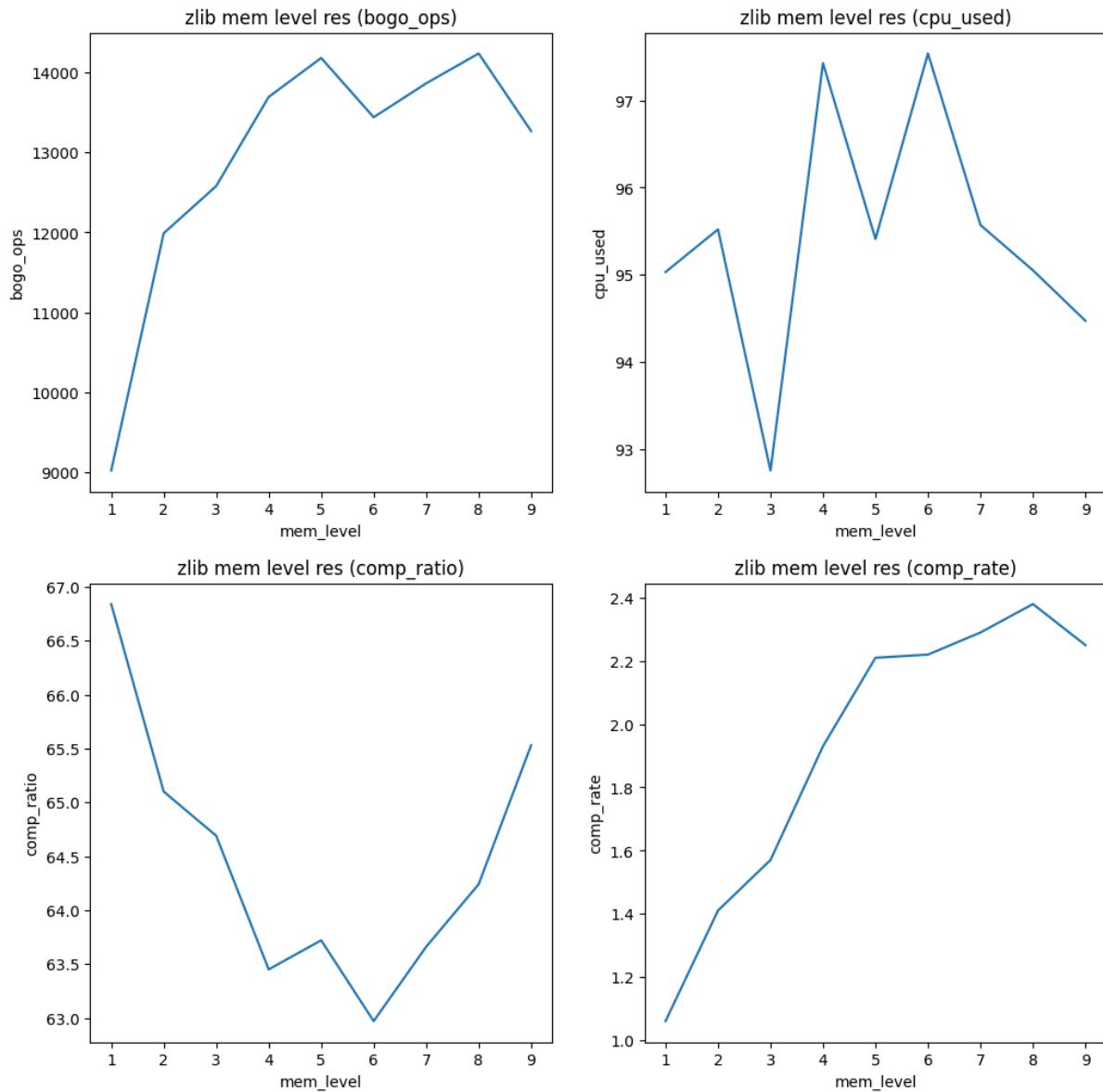
Memory

zlib-mem-level (specify zlib compression state memory usage)

Задает объем памяти, используемый библиотекой zlib, используемый для осуществления сжатия.

Тестовый скрипт: /memory/zlib-mem-level

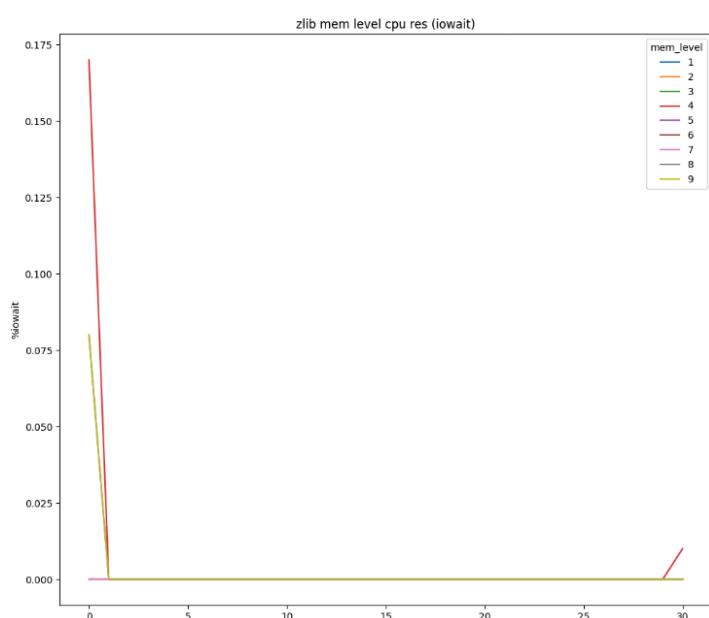
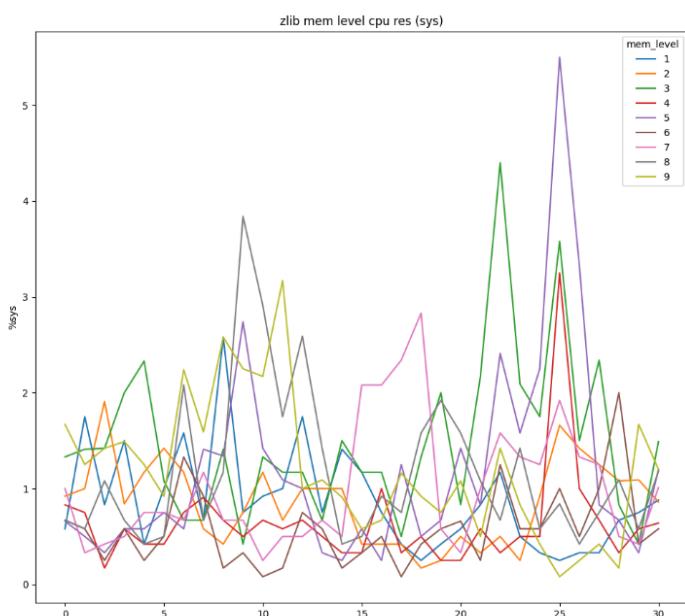
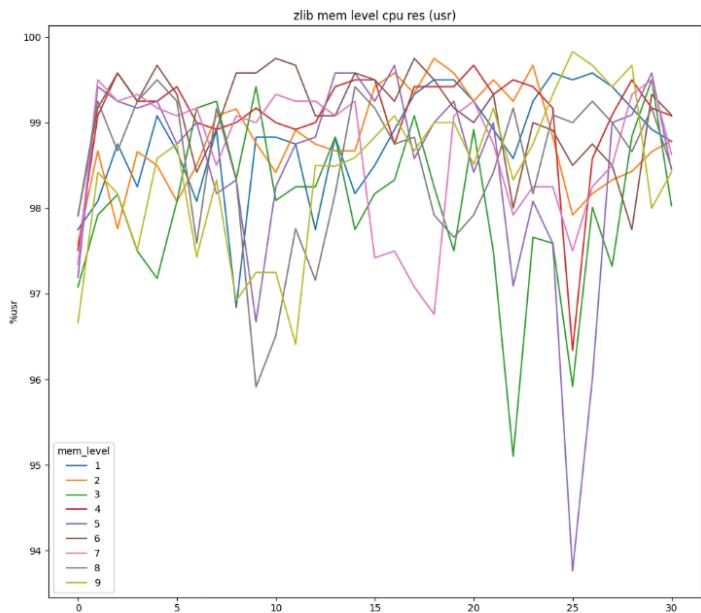
Измерим основные параметры (bogo ops, cpu used, compression ratio, compression rate), при различных значениях zlib-mem-level (допустимые значения [1;9]).



Лучший результат сжатия получен при наименьшем значении параметра, но при этом же значении получено наихудшее время сжатия.

Суммарная метрика bogo ops имеет наилучшие показатели при больших значениях параметра (>4).

Посмотрим на загрузку процессора:



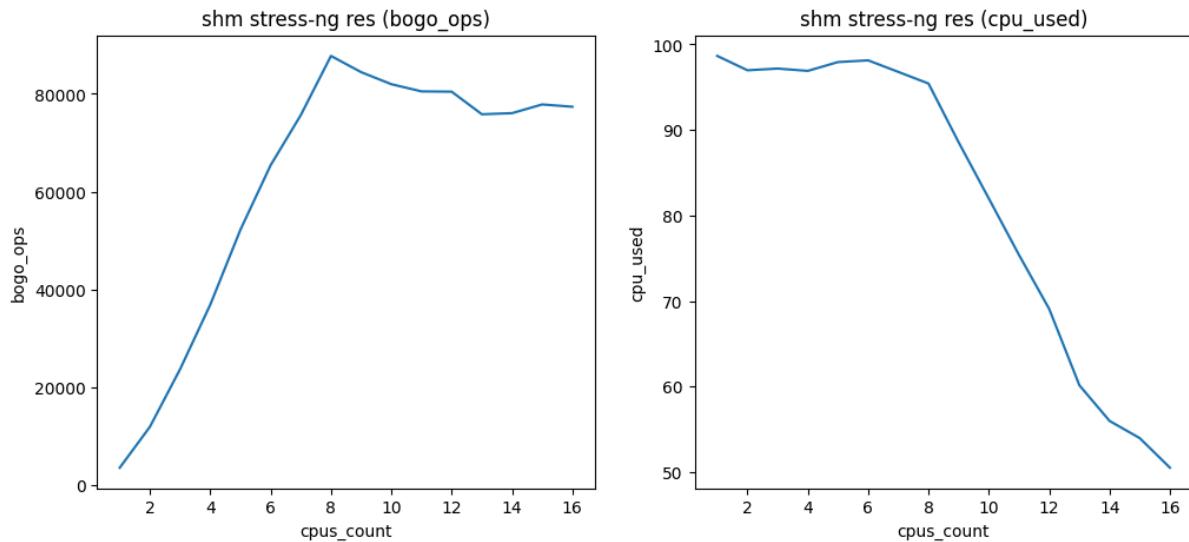
Как и предполагалось, большую часть времени процессор занят пользовательскими задачами.

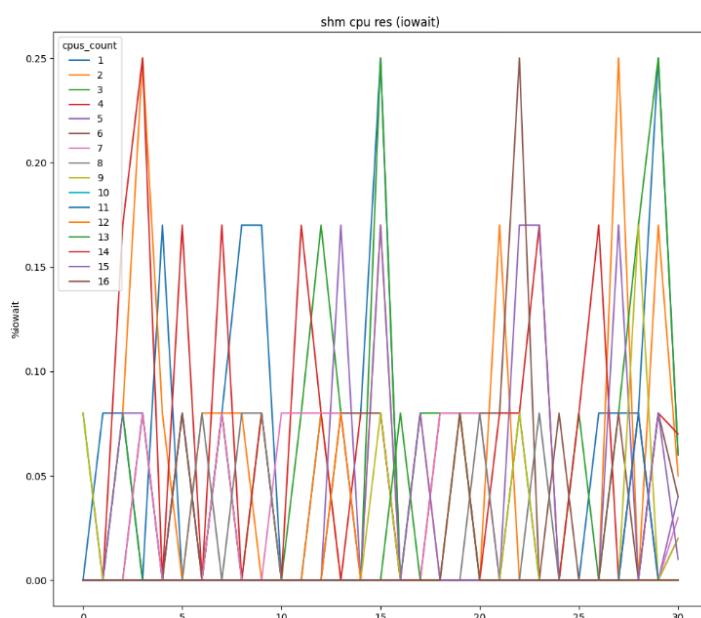
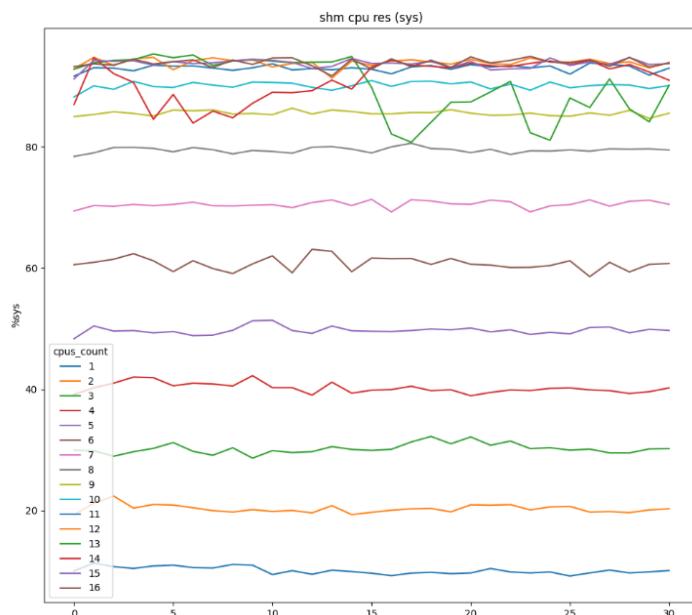
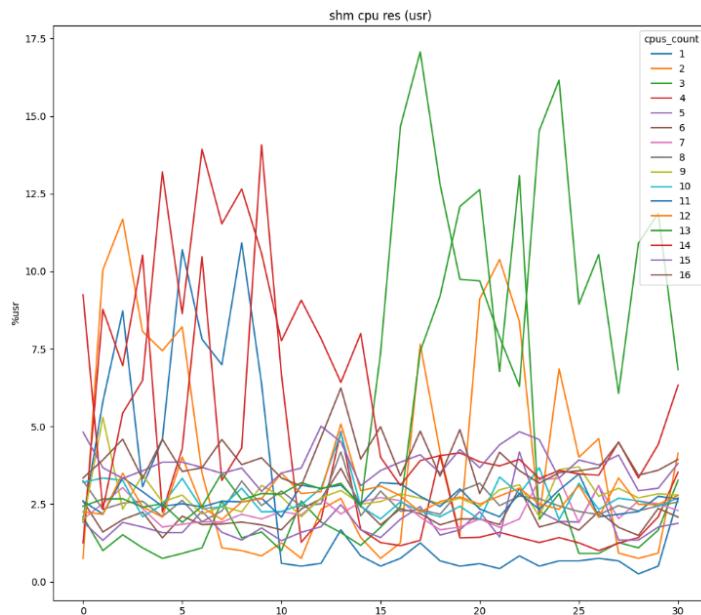
shm (start N workers that exercise POSIX shared memory)

Запускает воркеров, использующих разделяемую память.

Тестовый скрипт: /memory/shm

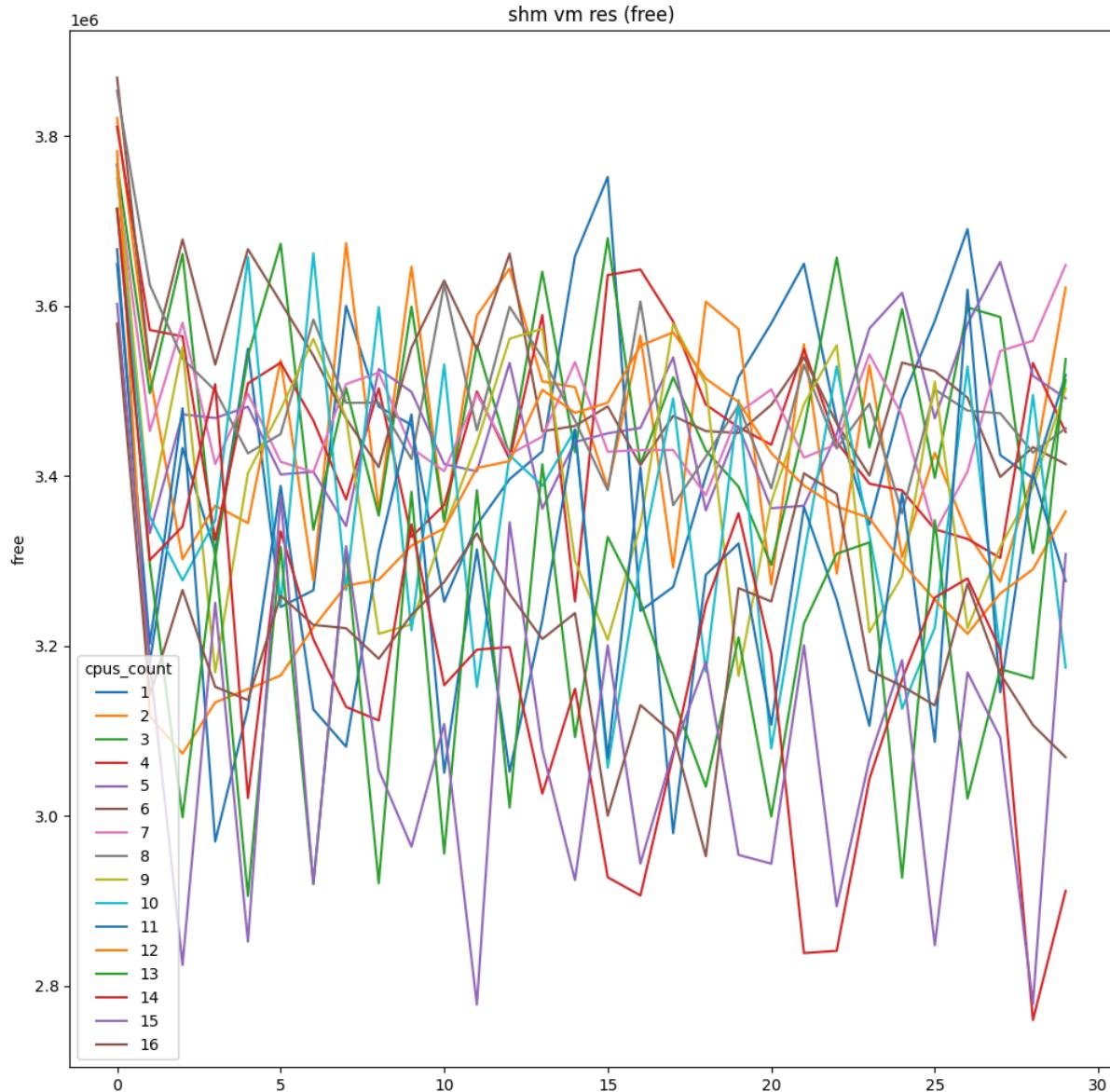
Стандартные тесты:





Видно, что процессор тратит большую часть времени в пространстве ядра, что, очевидно, связано со спецификой теста (частое использование системных вызовов для работы с виртуальной памятью).

Интересно дополнительно протестировать подсистему виртуальной памяти.



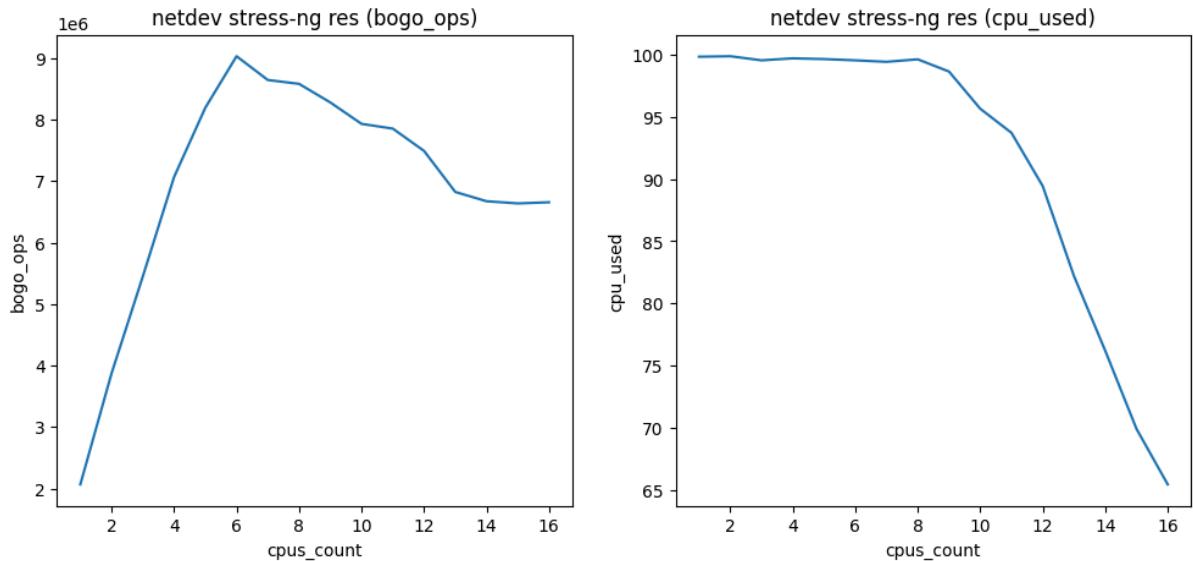
Свободной памяти меньше, чем в нормальном состоянии системы (без запущенного бенчмарка), значит все хорошо.

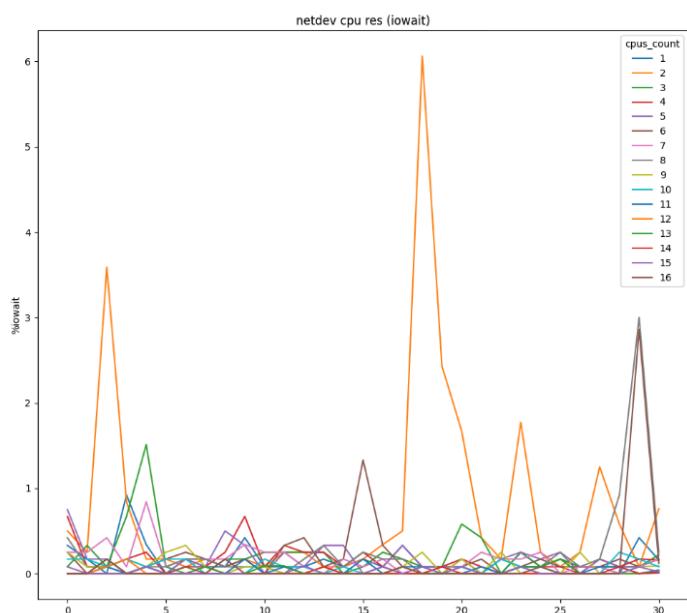
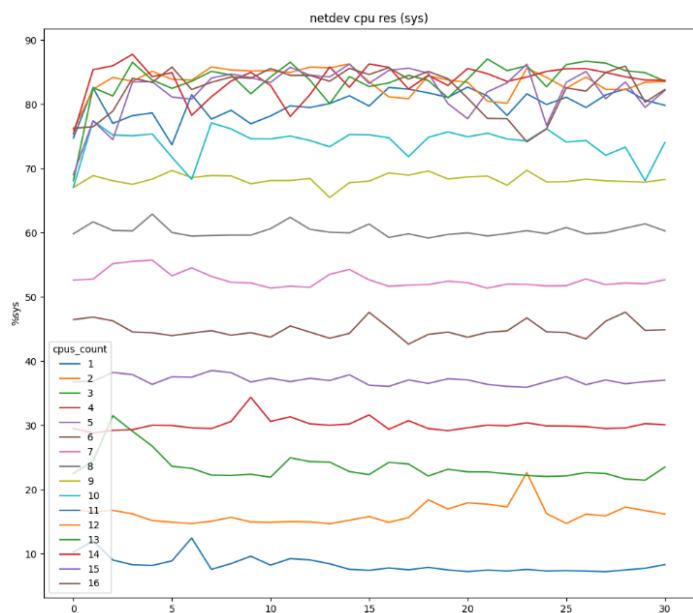
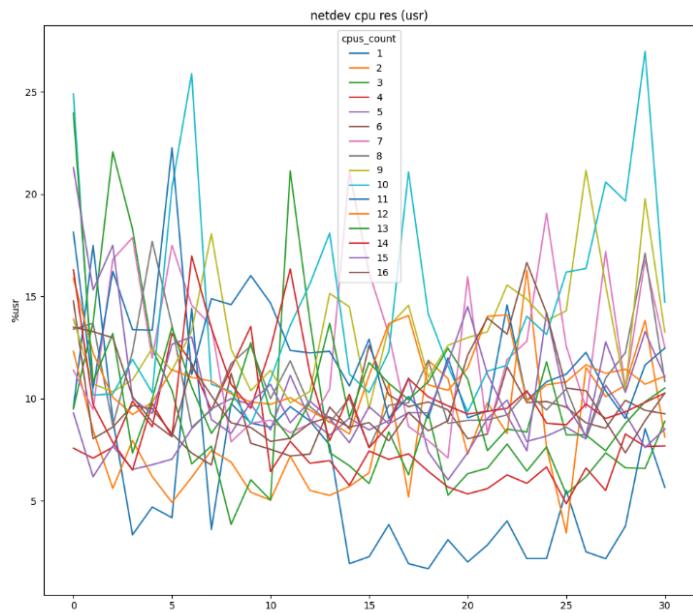
Network

netdev (start N workers exercising netdevice ioctls)

Запускает воркеров, взаимодействующих с параметрами ввода-вывода сетевых устройств.

Тестовый скрипт: /network/netdev





Опять видим высокий %sys, что, вероятно, связано с обилием системных вызовов. Можно на них посмотреть.

Найдем pid одного из воркеров с помощью утилиты top, а далее:

```
sudo strace -p <pid>
```

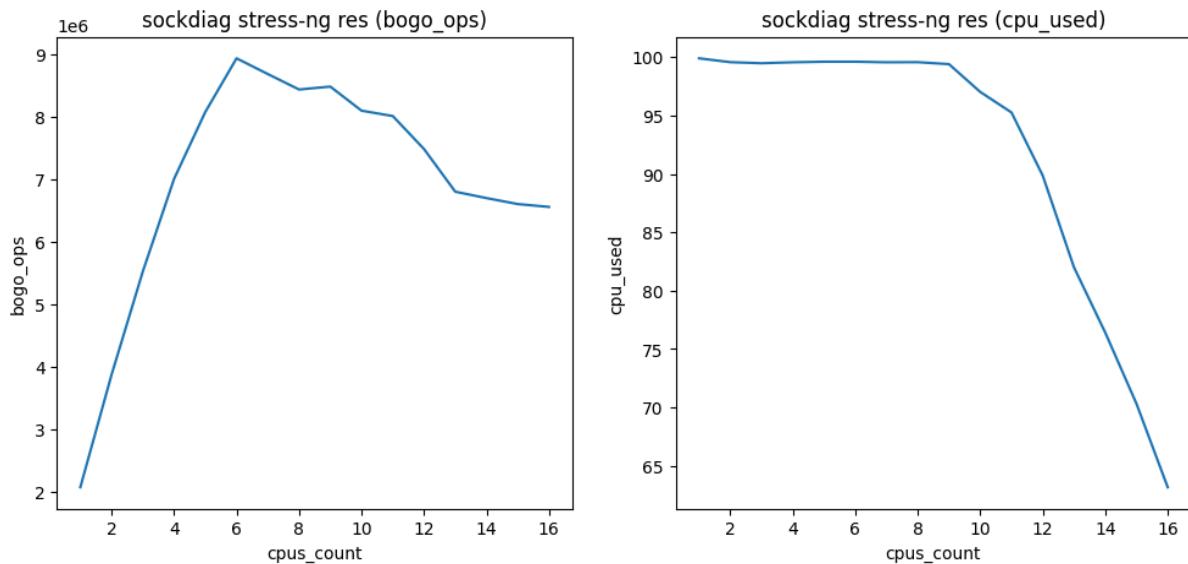
Получаем последовательности системных вызовов следующего вида:

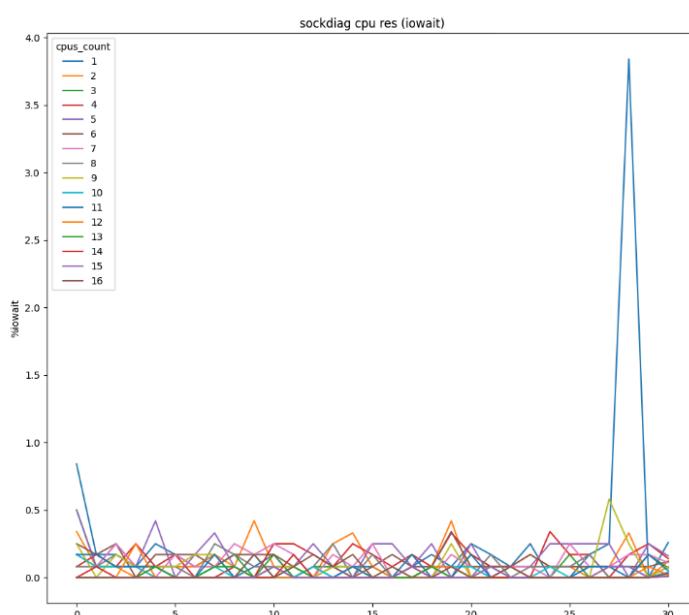
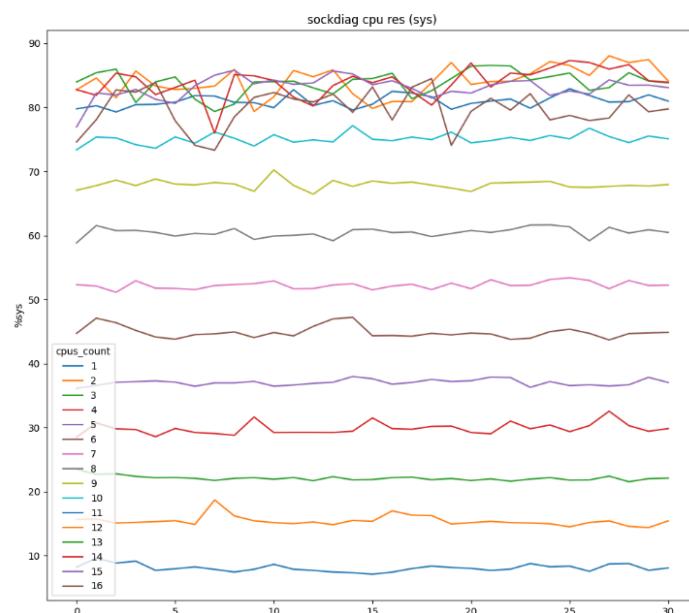
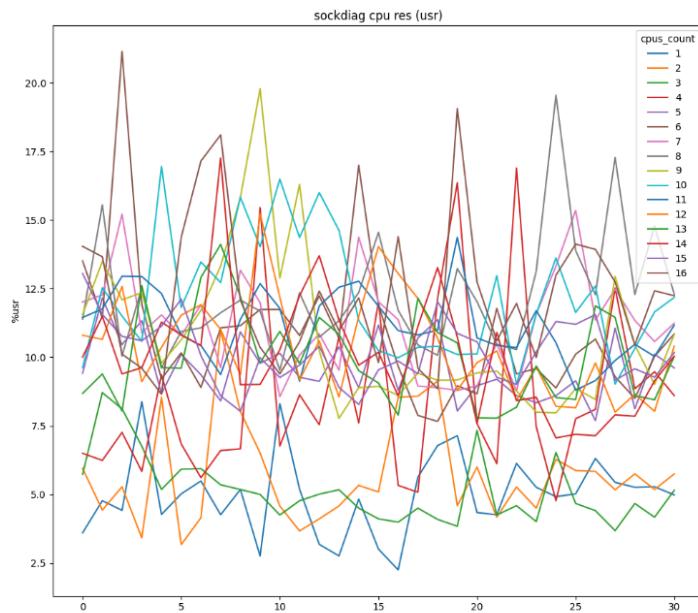
```
ioctl(4, SIOCGIFMEM, 0x564fc9d7c60)      = -1 ENOTTY (Inappropriate ioctl for device)
ioctl(4, SIOCGIFNAME, {ifr_ifindex=-1631756981}) = -1 ENODEV (No such device)
ioctl(4, SIOCGIFINDEX, {ifr_name="vmnet8", ifr_ifindex=4}) = 0
ioctl(4, SIOCGIFNAME, {ifr_ifindex=3, ifr_name="vmnet1"}) = 0
ioctl(4, SIOCGIFFLAGS, {ifr_name="vmnet1", ifr_flags=IFF_UP|IFF_BROADCAST|IFF_RUNNING|IFF_MULTICAST}) = 0
ioctl(4, SIOCGIFFLAGS, 0x564fc9d7c88) = -1 EINVAL (Invalid argument)
ioctl(4, SIOCGIFADDR, {ifr_name="vmnet1", ifr_addr={sa_family=AF_INET, sin_port=htons(0), sin_addr=inet_addr("172.16.35.1")}}) = 0
ioctl(4, SIOCGIFNETMASK, {ifr_name="vmnet1", ifr_netmask={sa_family=AF_INET, sin_port=htons(0), sin_addr=inet_addr("255.255.255.0")}}) = 0
ioctl(4, SIOCGIFMETRIC, {ifr_name="vmnet1", ifr_metric=0}) = 0
ioctl(4, SIOCGIFMTU, {ifr_name="vmnet1", ifr_mtu=1500}) = 0
ioctl(4, SIOCGIFHWADDR, {ifr_name="vmnet1", ifr_hwaddr={sa_family=ARPHRD_ETHER, sa_data=00:50:56:c0:00:01}}) = 0
ioctl(4, SIOCGIFMAP, {ifr_name="vmnet1", ifr_map={mem_start=0, mem_end=0, base_addr=0, irq=0, dma=0, port=0}}) = 0
ioctl(4, SIOCGIFTXQLEN, {ifr_name="vmnet1", ifr_qlen=1000}) = 0
ioctl(4, SIOCGIFSTADDR, {ifr_name="vmnet1", ifr_dstaddr={sa_family=AF_INET, sin_port=htons(0), sin_addr=inet_addr("172.16.35.1")}}) = 0
ioctl(4, SIOCGIFBRDADDR, {ifr_name="vmnet1", ifr_broadaddr={sa_family=AF_INET, sin_port=htons(0), sin_addr=inet_addr("172.16.35.255")}}) = 0
```

sockdiag (start N workers exercising sockdiag netlink)

Создает воркеров, совершающих системные вызовы `sock_diag`, для получения информации о сокетах.

Тестовый скрипт: `/network/sockdiag`





Получили похожую на прошлый тест картину. Процессор опять в основном занимается обработкой системных вызовов.

Тут совсем ничего не понятно, но можно разглядеть, что это действительно некая информация о сокетах.

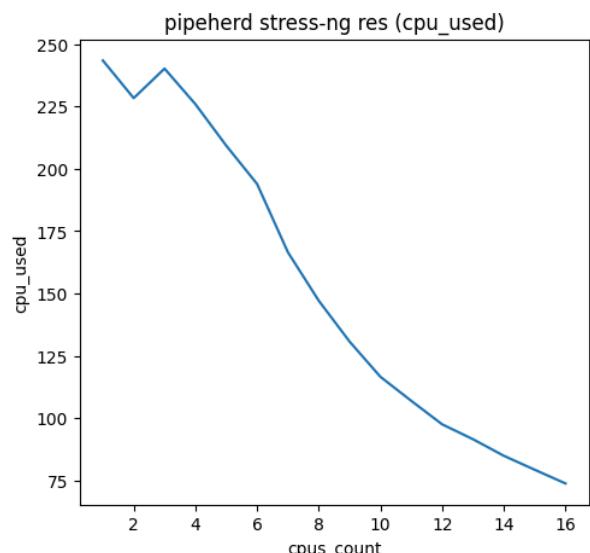
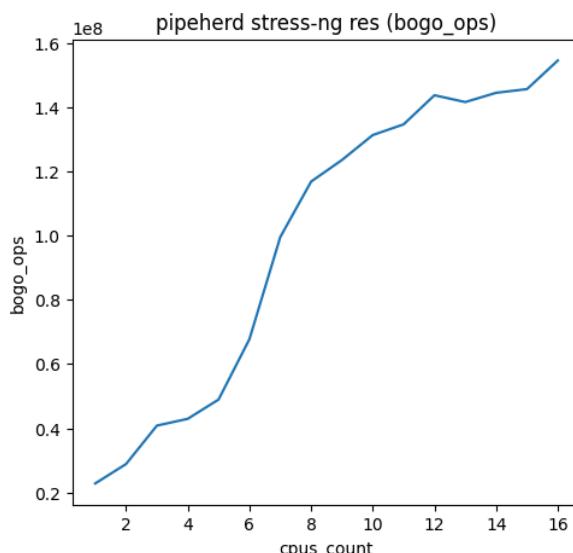
Pipe

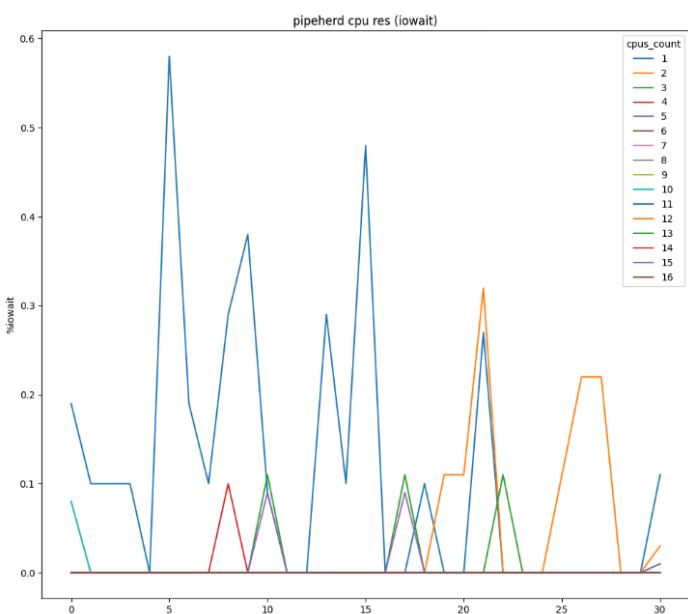
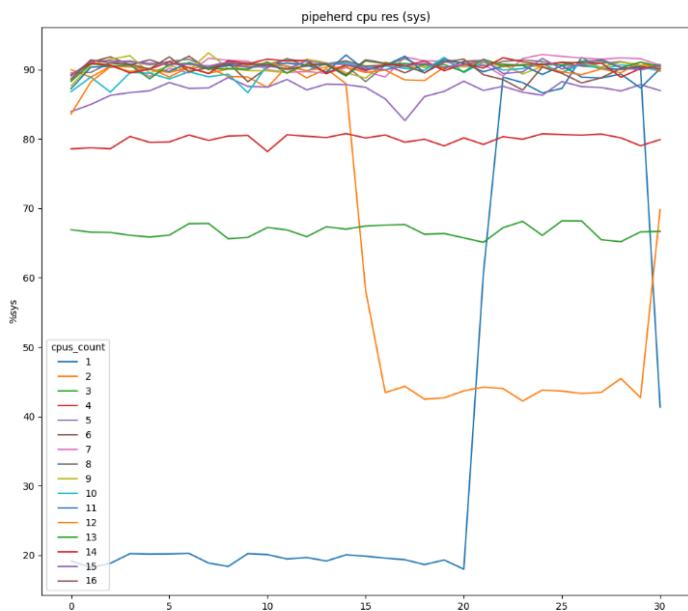
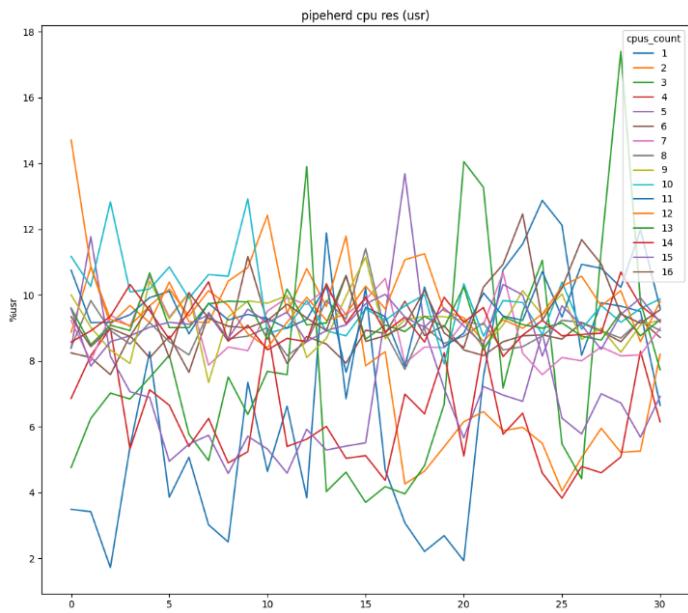
pipeherd (start N multi-process workers exercising pipes)

Создает воркеры, каждый из которых создает много процессов, взаимодействующих посредством пайпов.

Тестовый скрипт: /pipe/pipeherd

Тест сильно нагружает систему.





Много времени тратится на системные задачи, что, вероятно, связано с постоянными переключениями контекста и системными вызовами для коммуникации через пайпы.

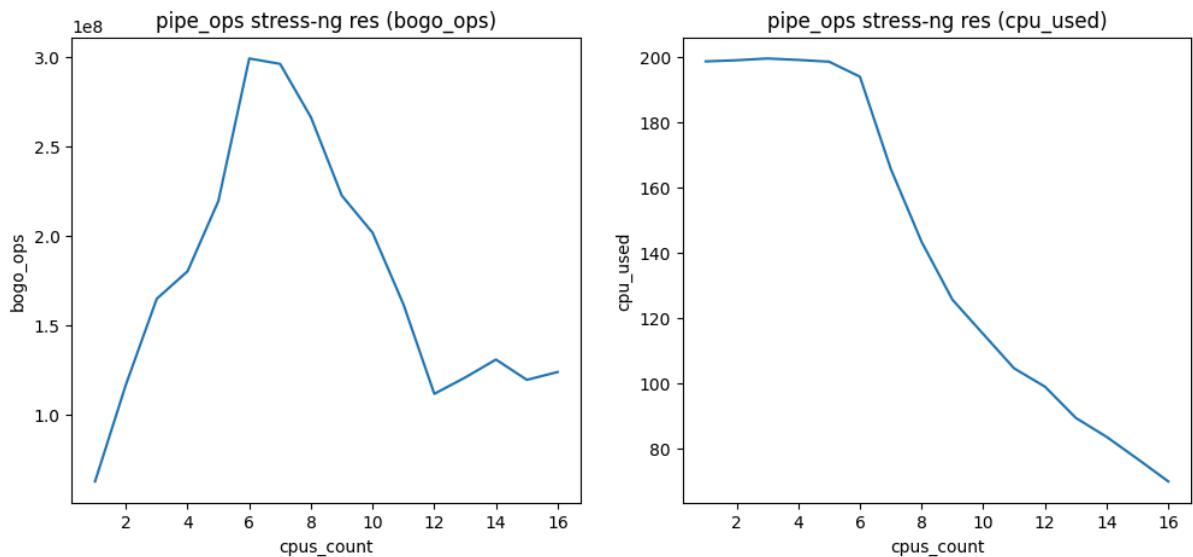
pipe-ops (stop after N pipe I/O bogo operations)

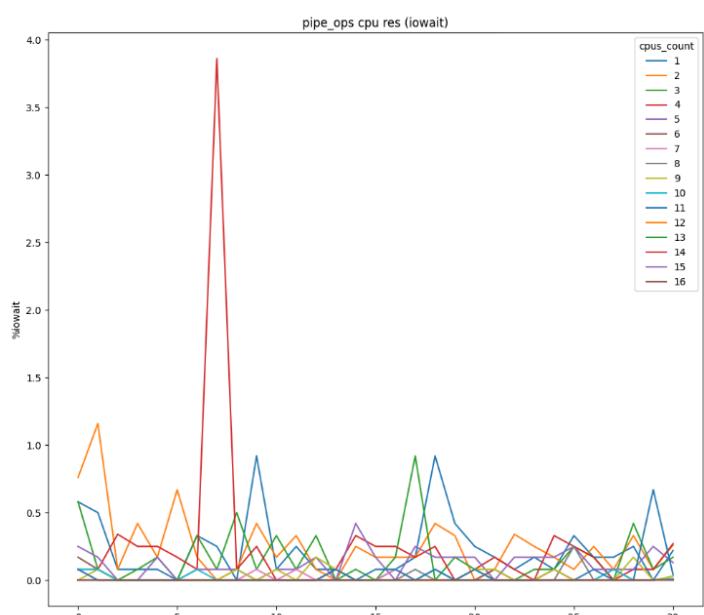
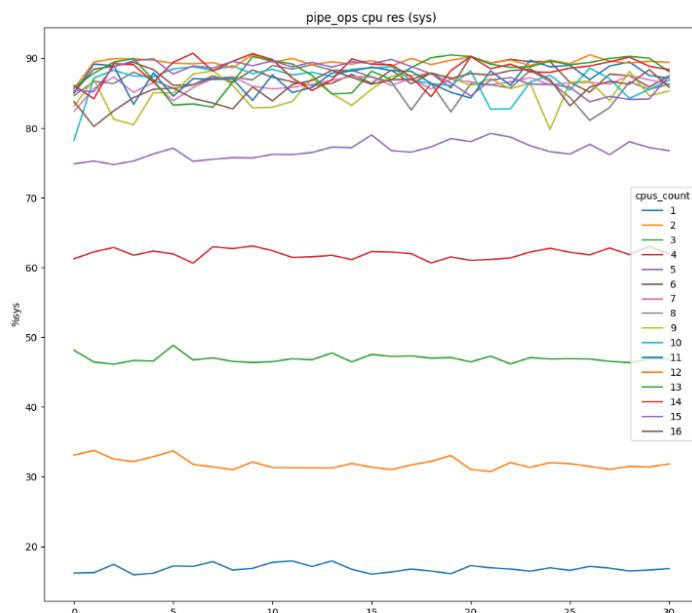
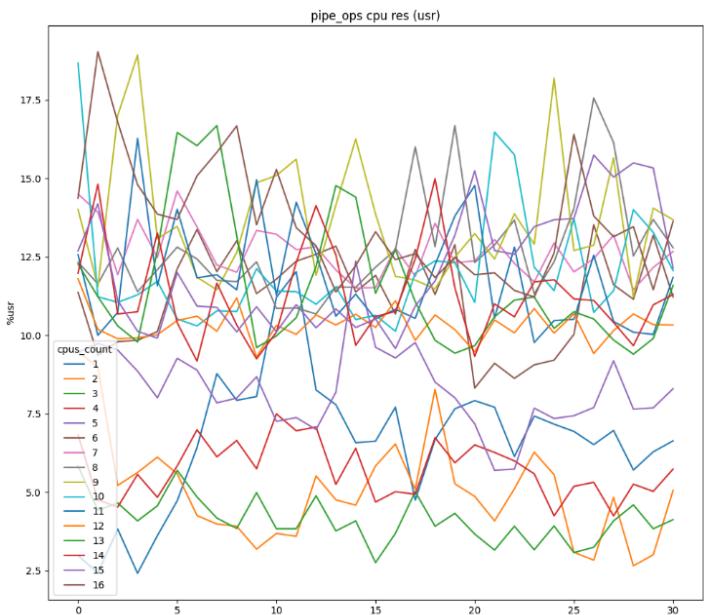
Позволяет остановить выполнение бенчмарка после достижения определенного количества bogo ops.

Тестовый скрипт: /pipe/pipe-ops

Не совсем ясно в чем смысл данного задания (этот параметр банально позволяет преждевременно остановить выполнение бенчмарка).

Просто протестируем --pipe.





Системные вызовы:

Картина аналогична предыдущему тесту.

Sched

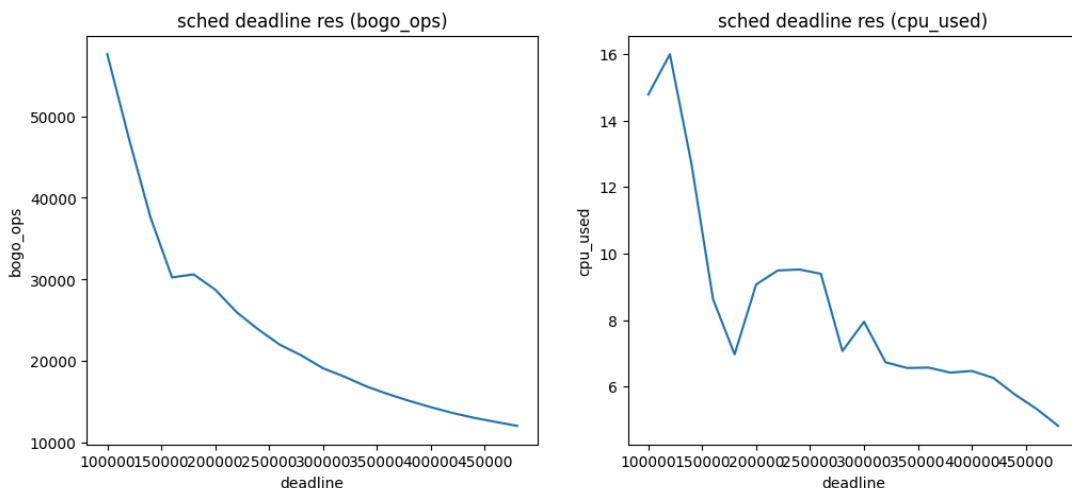
sched-deadline (set deadline for SCHED_DEADLINE to N nanosecs)

Устанавливает значение дедлайна для соответствующего алгоритма планирования.

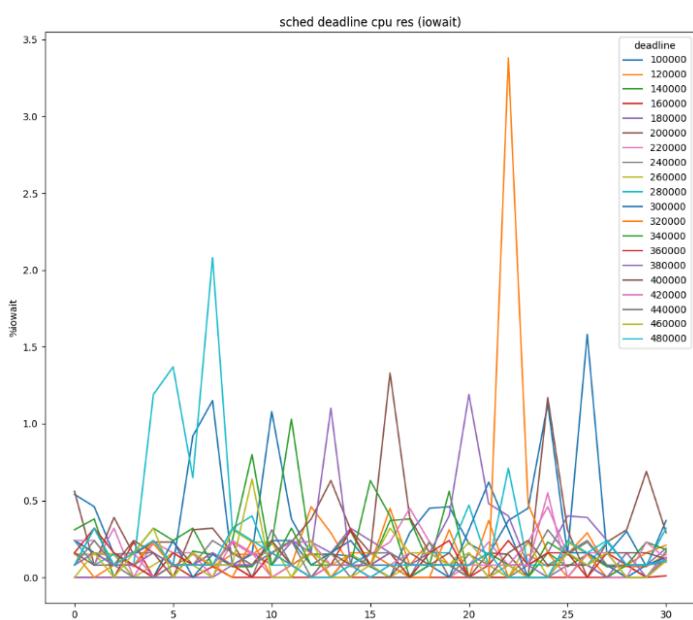
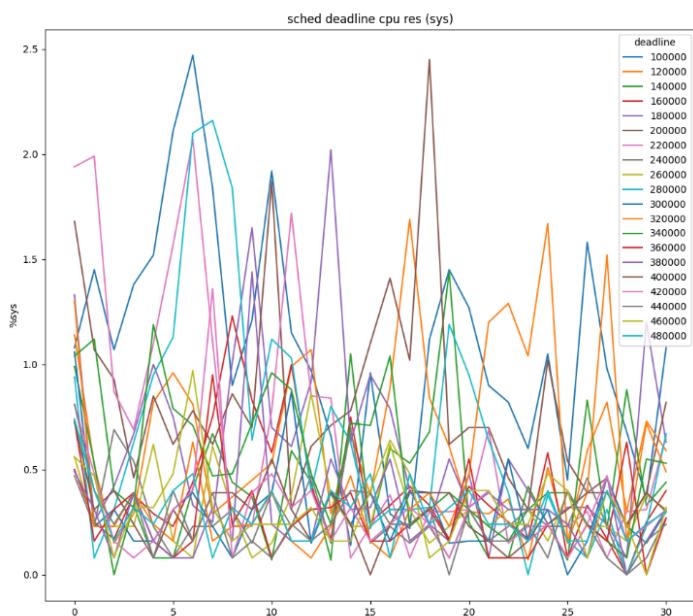
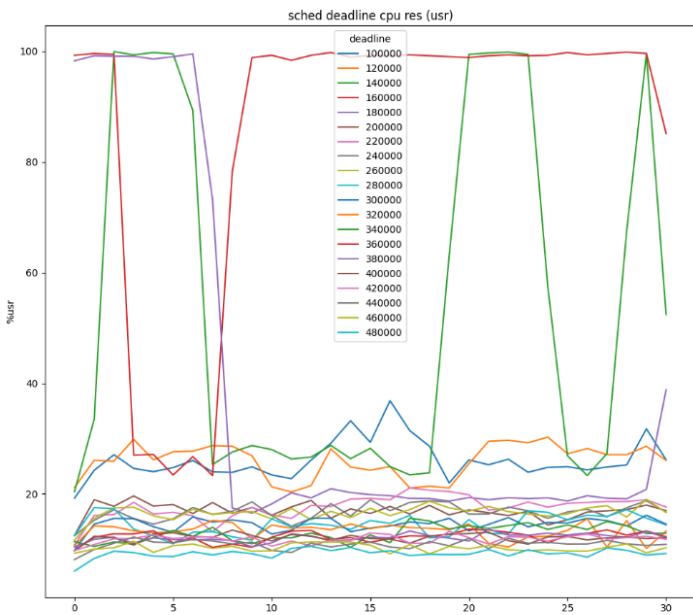
Тестовый скрипт: /sched/sched-deadline

Допустимые значения ≥ 100000 ns. Переберем $[100000, 500000]$ с шагом 20000.

```
sudo stress-ng --cpu 0 --sched deadline --sched-deadline $i  
--metrics --timeout $timeout
```



Ясно заметно снижение результатов метрики при увеличении значений параметра.



Процессор загружен не слишком сильно.

Некоторые выбросы в %usr, вероятно, связаны со сторонними процессами в системе.

sched-prio (set scheduler priority level N)

Устанавливает значение приоритета воркеров для соответствующего алгоритма планирования.

Тестовый скрипт: /sched/sched-prio

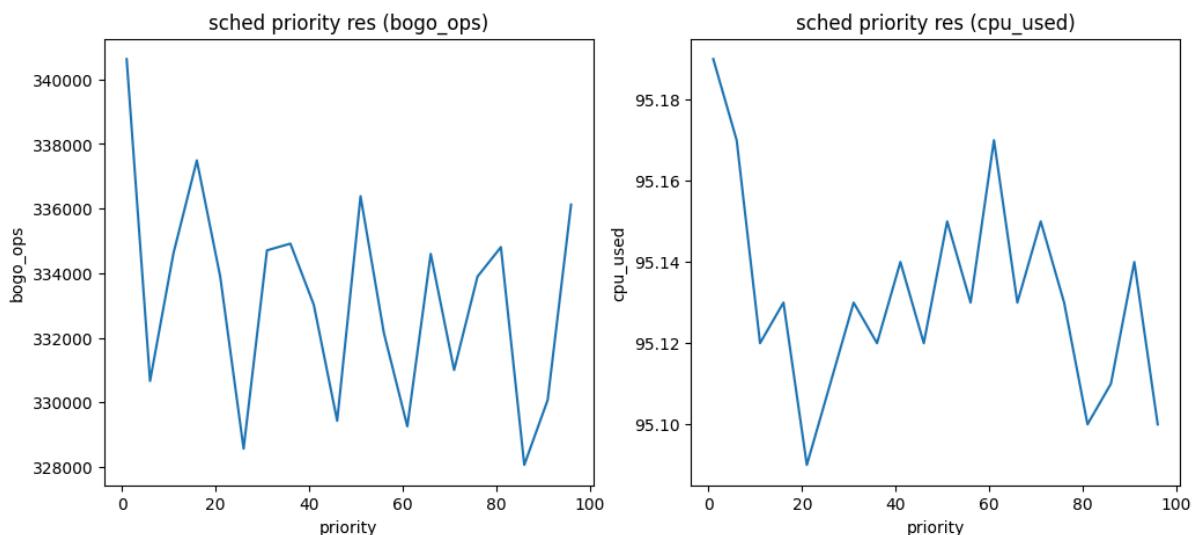
Для тестирования будем использовать алгоритм планирования round-robin.

Возможные значения приоритета [1,99].

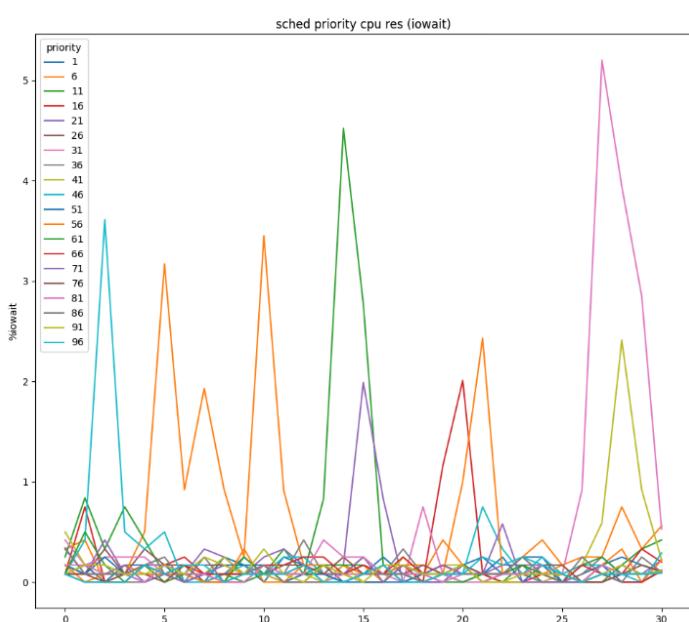
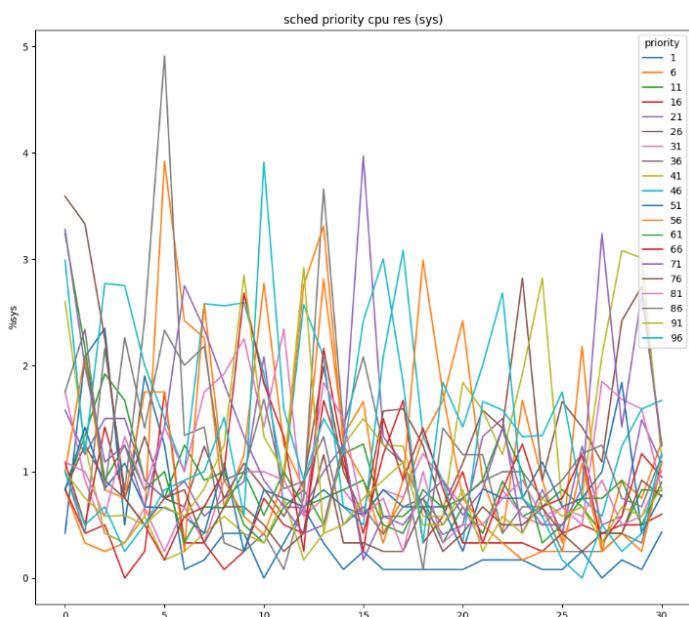
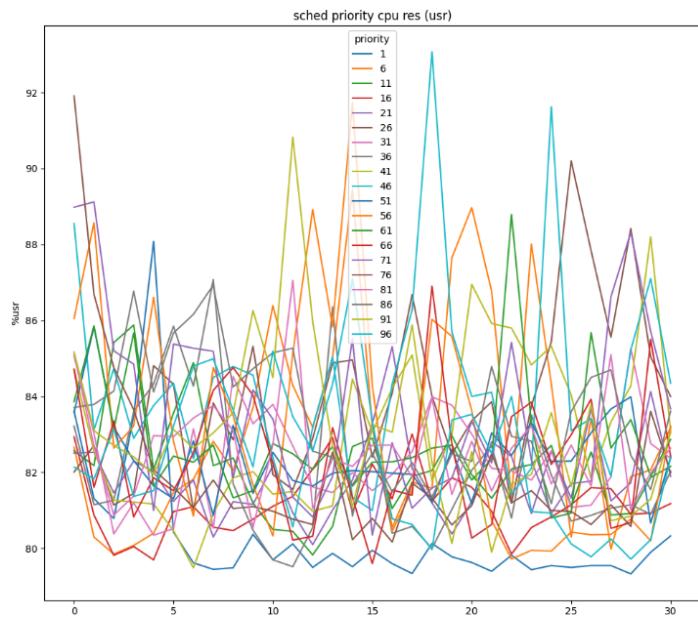
Переберем [1,99] с шагом 5.

Запустим 10 воркеров, так как иначе системе становится плохо, потому что высокоприоритетные воркеры занимают все доступные ядра.

```
sudo stress-ng --cpu 10 --sched rr --sched-prio $i --metrics  
--timeout $timeout
```



Не наблюдается значительного влияния параметра на результаты. Вероятно, это связано с тем, что даже с минимальным приоритетом воркерам хватает процессорного времени, так как фоновая нагрузка системы невысока.



Графики, соответствующие различным приоритетам, очень похожи, что подтверждает наш тезис.

Вывод

Выполняя эту лабораторную работу, я на практике познакомился с инструментами мониторинга и анализа производительности и поведения программ, воспользовался инструментом stress-ng, провел нагрузочное тестирование своей системы.