源码级别理解一下fio的使用 —— fio源码分析

- ref
    - https://fio.readthedocs.io/en/latest/fio_doc.html#interpreting-the-output
        - 官方文档，最权威的描述

- 摆一个使用例子
    - `fio -filename=/dev/sdb -direct=1 -zero_buffers=1 -iodepth=16 -thread -rw=randwrite -ioengine=psync -bs=4k -size=100G -numjobs=4 -runtime=3`
- 再摆两个统计的结果
    - **intel SATA ssd**

```
# 首先是异步IO
# fio -filename=/dev/sdb -direct=1 -zero_buffers=1 -iodepth=1 -thread -rw=randwrite -ioengine=libaio -bs=4k -size=100G -numjobs=1 -runtime=30 -gr
mytest: (g=0): rw=randwrite, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=libaio, iodepth=1
fio-3.16
Starting 1 thread
Jobs: 1 (f=1): [w(1)][100.0%][w=111MiB/s][w=28.3k IOPS][eta 00m:00s]
mytest: (groupid=0, jobs=1): err= 0: pid=6147: Sun Mar 19 15:31:31 2023
  write: IOPS=28.0k, BW=113MiB/s (119MB/s)(3398MiB/30001msec); 0 zone resets
    slat (nsec): min=1031, max=100349, avg=1684.98, stdev=551.66
    clat (nsec): min=1083, max=2308.8k, avg=32510.78, stdev=11716.18
     lat (usec): min=29, max=2309, avg=34.22, stdev=11.75
    clat percentiles (nsec):
     |  1.00th=[28544],  5.00th=[28544], 10.00th=[28800], 20.00th=[28800],
     | 30.00th=[29056], 40.00th=[29056], 50.00th=[29312], 60.00th=[29568],
     | 70.00th=[29568], 80.00th=[29824], 90.00th=[46848], 95.00th=[61696],
     | 99.00th=[63232], 99.50th=[66048], 99.90th=[69120], 99.95th=[71168],
     | 99.99th=[81408]
   bw (  KiB/s): min=112862, max=120088, per=100.00%, avg=116029.00, stdev=2195.42, samples=59
   iops        : min=28215, max=30022, avg=29007.22, stdev=548.85, samples=59
  lat (usec)   : 2=0.01%, 50=93.67%, 100=6.32%, 250=0.01%, 500=0.01%
  lat (msec)   : 2=0.01%, 4=0.01%
  cpu          : usr=1.83%, sys=7.48%, ctx=869998, majf=0, minf=1
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,869980,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
  WRITE: bw=113MiB/s (119MB/s), 113MiB/s-113MiB/s (119MB/s-119MB/s), io=3398MiB (3563MB), run=30001-30001msec

Disk stats (read/write):
  sdb: ios=52/867102, merge=0/0, ticks=9/28009, in_queue=28017, util=99.77%
###
###
# 再来个同步IO
# fio -filename=/dev/sdb -direct=1 -zero_buffers=1 -iodepth=1 -thread -rw=randwrite -ioengine=psync -bs=4k -size=100G -numjobs=1 -runtime=30 -gro
mytest: (g=0): rw=randwrite, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=psync, iodepth=1
fio-3.16
Starting 1 thread
Jobs: 1 (f=1): [w(1)][100.0%][w=116MiB/s][w=29.6k IOPS][eta 00m:00s]
mytest: (groupid=0, jobs=1): err= 0: pid=6182: Sun Mar 19 15:34:14 2023
  write: IOPS=28.8k, BW=113MiB/s (118MB/s)(3379MiB/30000msec); 0 zone resets
    clat (usec): min=29, max=1432, avg=34.44, stdev=10.03
     lat (usec): min=29, max=1432, avg=34.47, stdev=10.03
    clat percentiles (nsec):
     |  1.00th=[29824],  5.00th=[30080], 10.00th=[30080], 20.00th=[30336],
     | 30.00th=[30592], 40.00th=[30592], 50.00th=[31872], 60.00th=[32128],
     | 70.00th=[32384], 80.00th=[33024], 90.00th=[48384], 95.00th=[63232],
     | 99.00th=[64256], 99.50th=[68096], 99.90th=[70144], 99.95th=[72192],
     | 99.99th=[82432]
   bw (  KiB/s): min=109864, max=118784, per=99.95%, avg=115264.12, stdev=2494.02, samples=59
   iops        : min=27466, max=29696, avg=28816.00, stdev=623.52, samples=59
  lat (usec)   : 50=92.33%, 100=7.66%, 250=0.01%, 1000=0.01%
  lat (msec)   : 2=0.01%
  cpu          : usr=1.29%, sys=4.52%, ctx=864990, majf=0, minf=0
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,864943,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
  WRITE: bw=113MiB/s (118MB/s), 113MiB/s-113MiB/s (118MB/s-118MB/s), io=3379MiB (3543MB), run=30000-30000msec

Disk stats (read/write):
  sdb: ios=52/861918, merge=0/0, ticks=9/28173, in_queue=28182, util=99.79%
```

- fio测出来的最小IO延迟大约为**29us**，平均值在**34us**的样子，队列深度为**1**，task数也等于**1**（**随着队列深度的增大，IOPS在增大，但是延迟在增大**）

```
### 其实最应该测的是完成同样的数据写操作，需要用多长时间
### 测试数据总量

# sudo fio -filename=/dev/sdb -direct=1 -rw=randwrite -bs=4k -size=1G -numjobs=1 -name=mytest
mytest: (g=0): rw=randwrite, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=psync, iodepth=1
fio-3.16
Starting 1 process
Jobs: 1 (f=1): [w(1)][88.9%][w=115MiB/s][w=29.5k IOPS][eta 00m:01s]
mytest: (groupid=0, jobs=1): err= 0: pid=6840: Sun Mar 19 15:56:37 2023
  write: IOPS=29.5k, BW=115MiB/s (121MB/s)(1024MiB/8893msec); 0 zone resets
    clat (usec): min=29, max=2250, avg=33.75, stdev=11.57
     lat (usec): min=29, max=2250, avg=33.78, stdev=11.57
    clat percentiles (nsec):
     |  1.00th=[29824],  5.00th=[30080], 10.00th=[30080], 20.00th=[30080],
     | 30.00th=[30336], 40.00th=[30336], 50.00th=[30336], 60.00th=[30592],
     | 70.00th=[30592], 80.00th=[31104], 90.00th=[48384], 95.00th=[63232],
     | 99.00th=[64256], 99.50th=[68096], 99.90th=[70144], 99.95th=[72192],
     | 99.99th=[96768]
   bw (  KiB/s): min=115960, max=118600, per=100.00%, avg=117924.94, stdev=780.05, samples=17
   iops        : min=28990, max=29650, avg=29481.24, stdev=195.01, samples=17
  lat (usec)   : 50=93.51%, 100=6.48%, 250=0.01%
  lat (msec)   : 2=0.01%, 4=0.01%
  cpu          : usr=0.93%, sys=3.77%, ctx=262150, majf=0, minf=12
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,262144,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
  WRITE: bw=115MiB/s (121MB/s), 115MiB/s-115MiB/s (121MB/s-121MB/s), io=1024MiB (1074MB), run=8893-8893msec

Disk stats (read/write):
  sdb: ios=52/255014, merge=0/0, ticks=10/8215, in_queue=8224, util=99.21%

# dd if=/dev/zero of=/dev/sdb bs=4k count=256k oflag=direct
262144+0 records in
262144+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 8.93922 s, 120 MB/s
```

- **4k的bs**，**1G**的总大小，总耗时约为**8.9s**，**fio**与**dd**的测试结果基本是**一致**的

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 92.70    0.314814         356       883           wait4
  4.97    0.016868       16868         1           shmdt
  1.05    0.003549          86        41           futex
  0.37    0.001259           1       893         1 clock_nanosleep
  0.31    0.001058           4       236           mmap
  0.15    0.000526           0       897       892 stat
  0.13    0.000452           6        75           mprotect
  0.08    0.000286           1       193        16 openat
  0.05    0.000177           3        47           munmap
  0.04    0.000150           6        22           clone
  0.04    0.000142           2        62           read
  0.04    0.000122           0       177           fstat
  0.03    0.000116           0       177           close
  0.01    0.000022           1        14           write
  0.01    0.000021           2         8           pread64
  0.00    0.000012           6         2         1 arch_prctl
  0.00    0.000010           1         9           brk
  0.00    0.000004           0         5           rt_sigaction
  0.00    0.000003           3         1           prlimit64
  0.00    0.000002           2         1           lseek
  0.00    0.000002           2         1           rt_sigprocmask
  0.00    0.000002           1         2         1 shmctl
  0.00    0.000002           2         1           set_tid_address
  0.00    0.000002           2         1           restart_syscall
  0.00    0.000002           2         1           set_robust_list
  0.00    0.000001           0         2           getpid
  0.00    0.000000           0       493           lstat
  0.00    0.000000           0         1           ioctl
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           shmget
  0.00    0.000000           0         1           shmat
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2           sysinfo
  0.00    0.000000           0         2           sched_getaffinity
  0.00    0.000000           0       242           getdents64
------ ----------- ----------- --------- --------- ----------------
100.00    0.339604                  4496       912 total
```

- strace尝试跟踪write系统调用，这个时间非常小的原因找到了，还是利用fio测试的时候反应过来的，下面是strace fio的结果
  - 其实最直观的感受就是**我草，为啥没有write系统调用**，反而是**wait系统调用**耗时最多，后来结合fio的代码结构，主线程创建出众多的工作线程之后开始调用wait
    - 所以strace是没有跟踪到目标进程额外创建出的进程或线程的，需要**单独strace**
- 所以strace如何跟踪**额外的工作线程**呢
  - `-o filename` —— 默认strace将结果输出到stdout
    - 通过 `-o` 可以将输出写入到filename文件中
  - `-ff` —— 常与 `-o` 选项一起使用，不同进程（**子进程**）产生的系统调用输出到 `filename.PID` 文件
    - `-ff` 与 `-c` 是矛盾的
    - `ff` means `--follow-forks`
  - `-f` —— 这个应该是最直观的
- 但是通过该方法可以确定的是dd是没有创建额外线程的，就是它自己，但是就它自己write的耗时也不是直观理解的样子
  - 如下所示，还是需要继续**找原因**
- 加上 `-f` 参数，可以查看进程内**所有线程**的数据读取，打开 `-T` 选项观察系统调用时长， `-tt` 显示跟踪时间

```
#...
17:24:11.020662 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000039>
17:24:11.020707 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000003>
17:24:11.020717 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000039>
17:24:11.020763 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000003>
17:24:11.020773 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000038>
17:24:11.020818 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000003>
17:24:11.020827 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000038>
17:24:11.020873 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000003>
17:24:11.020882 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000039>
17:24:11.020928 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000003>
17:24:11.020938 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000038>
17:24:11.020983 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000003>
17:24:11.020992 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000038>
17:24:11.021038 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000003>
17:24:11.021047 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000038>
17:24:11.021093 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000003>
17:24:11.021103 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096 <0.000038>
#...
```

- 这里所展示出的**read/write系统调用**所用的时间是符合预期的，读操作作用的是/dev/zero，所以系统调用仅为3us，而write是要真的落盘的，实际调用时间为 `30-40us` ，其实还有大量没有截取到的，个别IO延迟将达到**百us级别**，只能说完全符合预期

```
zc@zc-System-Product-Name:~/workspace/tmp$ sudo strace -c dd if=/dev/zero of=/dev/sdb bs=4k count=256k oflag=direct
262144+0 records in
262144+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 14.2685 s, 75.3 MB/s
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 61.41    0.602536           2    262147           write
 38.59    0.378652           1    262147           read
  0.00    0.000024           1        18        11 openat
  0.00    0.000010           1        10           close
  0.00    0.000003           0         9           mmap
  0.00    0.000001           0         5           fstat
  0.00    0.000000           0         1           lseek
  0.00    0.000000           0         3           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         3           rt_sigaction
  0.00    0.000000           0         6           pread64
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         2           dup2
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2         1 arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.981226                524359        13 total
```

- 那么最后一个问题是什么呢，strace的 `-c` 到底**统计**的是什么，在官网上找到了答案
  - ref
    - https://man7.org/linux/man-pages/man1/strace.1.html
  - This attempts to show **system time** (**CPU time spent running in the kernel**) independent of **wall clock time**
    - 下文详细解释 **CPU time** and **wall clock time**
  - 那这个就是CPU跑内核代码的时间
    - 所以在这个IO情景下，总耗时约为**9s**左右，而系统的CPU时间在**0.9s**左右，即软件栈的时间，所以对于磁盘来说，软件栈的开销约为整体IO时间的**10%**
      - 至少在靠近理论预期了，虽然**比例还有点大**
- 那么什么是**CPU time**
  - https://zhuanlan.zhihu.com/p/39891521
  - https://en.wikipedia.org/wiki/CPU_time —— https://en.wikipedia.org/wiki/Elapsed_real_time —— 解释了什么是 **CPU time/wall clock time**
    - 概念解析
      - CPU time (or process time) is the amount of time for which a central processing unit (CPU) was used for processing instructions of a computer program or operating system, **as opposed to elapsed time**, which includes for example, waiting for input/output (I/O) operations or entering low-power (idle) mode
        - 纯执行指令的时间，不包括等待IO的时间
      - In contrast, elapsed real time (or simply real time, or wall-clock time) is the time taken from the start of a computer program until the end as measured by an ordinary clock. Elapsed real time includes I/O time, any multitasking delays, and all other types of waits incurred by

    the program
      - **wall-clock time**其实就是整体时间，包括了IO的时间
  - Subdivision
    - CPU time or CPU usage can be reported either for each thread, for each process or for the entire system. Moreover, depending on what exactly the CPU was doing, the reported values can be subdivided in
      - **User time** is the amount of time the CPU was busy executing code in **user space**
        - **执行用户态代码的时间**
      - **System time** is the amount of time the CPU was busy executing code in **kernel space**
        - 执行**内核态代码**的时间，不包括IO的等待时间
      - **Idle time** (for the whole system only) is the amount of time the CPU was not busy, or, otherwise, the amount of time it executed the System Idle process. Idle time actually measures unused CPU capacity
      - **Steal（窃取）time** (for the whole system only), on **virtualized hardware**, is the amount of time the operating system wanted to execute, but was not allowed to by the hypervisor. This can happen if the physical hardware runs multiple guest operating system and the hypervisor chose to allocate a CPU time slot to another one

fio源码解析

- fio本身**框架**是比较清晰的，主线程解析用户参数，然后创建用户需要的**工作线程执行IO操作**，但是在用 `strace -f` 的过程中发现，假设 `num_jobs=1` ，fio可不止起2个线程，比想象中还要多

fio源码分析

- `int main(int argc, char *argv[], char *envp[])` —— `fio/fio.c`
  - `initialize_fio(envp)` —— 这里与具体的体系结构**息息相关**，可以理解为**运行环境**的初始化
    - `err = endian_check();`
    - `arch_init(envp);`
  - `parse_options(argc, argv);` —— **参数解析**，就是我们**fio**输入的那**一票参数**
    - `fio_init_options();`
    - `job_files = parse_cmd_line(argc, argv, type);`
      - `ioengine_load(td);`
        - `td->io_ops = load_ioengine(td);`
          - `ops = __load_ioengine(td->o.ioengine);`
            - `return find_ioengine(engine);`
      - `ret = fio_cmd_option_parse(td, opt, val);`
        - `ret = parse_cmd_option(opt, val, fio_options, &td->o, &td->opt_list);`
          - `handle_option(o, val, data);`
            - `do {`
              - `__ret = __handle_option(o, ptr, data, !done, !!ptr2, done);`
                - `struct fio_option o`
                  - 这是最关键的一个数据结构，用户fio后面跟着的所有参数被解析后都会保存到这个数据结构中
                    - **以供工作线程使用**
            - `} while (1);`
  - `fio_time_init();` —— 有一个比较复杂的**时钟校准**过程
    - `fio_clock_init();`
      - `calibrate_cpu_clock()`
        - `get_cycles_per_msec();`
  - `fio_backend(NULL);` —— **fio工作函数**
    - `helper_thread_create(startup_sem, sk_out)` —— 创建**计时线程**
      - `ret = pthread_create(&hd->thread, NULL, helper_thread_main, hd);`
        - `...`
    - `run_threads(sk_out);` —— 创建**工作线程**，所有的**IO**都是在这里创建，在这里**结束**
      - `for_each_td(td) {`
        - `fd = calloc(1, sizeof(*fd));`
        - `fd->td = td;`
        - `if (td->o.use_thread)` —— **多线程**模式

      - `ret = pthread_create(&td->thread, NULL, thread_main, fd);`
        - **thread_main 详见下文**
      - `ret = pthread_detach(td->thread);`
    - `else` —— **多进程**模式
      - `pid = fork();`
      - `if (!pid)` —— **子进程**
        - `ret = (int)(uintptr_t)thread_main(fd);`
          - **详见下文**
  - `} end_for_each();`
  - `while (nr_running)`
    - `reap_threads(&nr_running, &t_rate, &m_rate);`
      - `for_each_td(td) {`
        - `ret = waitpid(td->pid, &status, flags);`
          - **主线程就会阻塞在对应的fd上，直到所有的工作线程都完成**
      - `} end_for_each();`
    - `do_usleep(10000);`
- `helper_thread_exit();`
- `if (!fio_abort)` —— **不出错就打印统计数据**
  - `__show_run_stats();`
    - `show_thread_status_normal(ts, rs, out);`
      - `show_ddir_status(rs, ts, ddir, out);`
      - `if (calc_lat(&ts->slat_stat[ddir], &min, &max, &mean, &dev))`
        - `display_lat("slat", min, max, mean, dev, out);`
      - `if (calc_lat(&ts->clat_stat[ddir], &min, &max, &mean, &dev))`
        - `display_lat("clat", min, max, mean, dev, out);`
      - `if (calc_lat(&ts->lat_stat[ddir], &min, &max, &mean, &dev))`
        - `display_lat(" lat", min, max, mean, dev, out);`

```
/* 异步引擎 */
FIO_STATIC struct ioengine_ops ioengine = {
        .name                   = "libaio",
        .version                = FIO_IOOPS_VERSION,
        .flags                  = FIO_ASYNCIO_SYNC_TRIM |
                                        FIO_ASYNCIO_SETS_ISSUE_TIME,
        .init                   = fio_libaio_init,
        .post_init              = fio_libaio_post_init,
        .prep                   = fio_libaio_prep,
        .queue                  = fio_libaio_queue,
        .commit                 = fio_libaio_commit,
        .cancel                 = fio_libaio_cancel,
        .getevents              = fio_libaio_getevents,
        .event                  = fio_libaio_event,
        .cleanup                = fio_libaio_cleanup,
        .open_file              = generic_open_file,
        .close_file             = generic_close_file,
        .get_file_size          = generic_get_file_size,
        .options                = options,
        .option_struct_size     = sizeof(struct libaio_options),
};
/* 同步引擎 */
static struct ioengine_ops ioengine_prw = {
        .name           = "psync",
        .version        = FIO_IOOPS_VERSION,
        .queue          = fio_psyncio_queue,
        .open_file      = generic_open_file,
        .close_file     = generic_close_file,
        .get_file_size  = generic_get_file_size,
        .flags          = FIO_SYNCIO,
};
/* 同步引擎的入队方法 */
static enum fio_q_status fio_psyncio_queue(struct thread_data *td,
                                           struct io_u *io_u)
{
        struct fio_file *f = io_u->file;
        int ret;

        fio_ro_check(td, io_u);
    /*
     * 这里其实可以补充一句pread与read的区别，由于lseek和read调用之间内核可能会临时挂起进程，所以对同步造成了问题
     * 调用pread相当于顺序调用了lseek和read，这两个操作相当于一个捆绑的原子操作
     */
        if (io_u->ddir == DDIR_READ)
                ret = pread(f->fd, io_u->xfer_buf, io_u->xfer_buflen, io_u->offset);
        else if (io_u->ddir == DDIR_WRITE)
                ret = pwrite(f->fd, io_u->xfer_buf, io_u->xfer_buflen, io_u->offset);
        else if (io_u->ddir == DDIR_TRIM) {
                do_io_u_trim(td, io_u);
                return FIO_Q_COMPLETED;
        } else
                ret = do_io_u_sync(td, io_u);

        return fio_io_end(td, io_u, ret);
}
```

- `thread_main` —— **主要的job入口** —— fio多线程或多进程的时候文件的打开位于线程或进程中，**即同一个文件会被打开多次**
    - `if (fio_option_is_set(o, cpumask))`
        - `if (o->cpus_allowed_policy == FIO_CPUS_SPLIT)`
            - `ret = fio_cpus_split(&o->cpumask, td->thread_number - 1);`
        - `ret = fio_setaffinity(td->pid, o->cpumask);`
    - `td_io_init(td);`
        - `if (td->io_ops->init)`
            - `ret = td->io_ops->init(td);` —— 异步io有init方法，但是同步io没有init方法，以**libaio**为例
                - **fio_libaio_init**，就是调用**libaio**的**io_setup**方法
    - `init_io_u(td);`
        - `max_units = td->o.iodepth;` —— 这是用户传入的**队列深度**
        - `for (i = 0; i < max_units; i++)` —— **追踪每一个IO**
            - `ptr = fio_memalign(cl_align, sizeof(*io_u), td_offload_overlap(td));`
            - `io_u = ptr;`
            - `memset(io_u, 0, sizeof(*io_u));` —— 每一个IO都对应一个 `io_unit` ，相当于handle

    - `io_u_qpush(&td->io_u_freelist, io_u);`
    - `io_u_qpush(&td->io_u_all, io_u);`
    - `if (td->io_ops->io_u_init)`
      - `int ret = td->io_ops->io_u_init(td, io_u);`
- `td->io_ops->post_init(td);` —— `libaio` 将调用 `fio_libaio_post_init` 方法
- `set_epoch_time(td, o->log_unix_epoch | o->log_alternate_epoch, o->log_alternate_epoch_clock_id);`
  - `fio_gettime(&td->epoch, NULL);` —— 这个是**总**的**开始时间**
- `fio_getrusage(&td->ru_start);`
  - `return getrusage(RUSAGE_SELF, ru);` —— **统计CPU利用率**
    - `h = GetCurrentProcess();`
    - `GetProcessTimes(h, &cTime, &eTime, &kTime, &uTime);`
- `do_io(td, bytes_done);`
  - `total_bytes = td->o.size;` —— 用户输入的size是每一个线程的工作size，**而不是总量**
  - `io_u = get_io_u(td);`
    - `io_u = __get_io_u(td);`
    - `if (!td->o.disable_lat)`
      - `fio_gettime(&io_u->start_time, NULL);` —— 每一个 `io_u` 对应的都是**独立的IO个体**，这里代表的是IO的**开始时间**
  - `ret = io_u_submit(td, io_u);`
    - `return td_io_queue(td, io_u);`
      - `ret = td->io_ops->queue(td, io_u);` —— **同步异步queue的方法是不同的，这里分别说明一下**
        - `static enum fio_q_status fio_psyncio_queue(struct thread_data *td, struct io_u *io_u)` —— **同步IO**的**queue方法**
          - `ret = pwrite(f->fd, io_u->xfer_buf, io_u->xfer_buflen, io_u->offset);`
            - psync的系统调用是pwrite，说白了就是write系统调用，一定是IO结束后才返回
          - `return fio_io_end(td, io_u, ret);`
            - `return FIO_Q_COMPLETED;`
        - `static enum fio_q_status fio_libaio_queue(struct thread_data *td, struct io_u *io_u)` —— **异步IO**的**queue方法**
          - `struct libaio_data *ld = td->io_ops_data;`
          - `ld->queued++;`
            - 对于异步IO而言，这里确实就仅仅是**入队**，这个是用户给的**iodepth**，一次性入队的IO数量
          - `return FIO_Q_QUEUED;`
      - 到此为止，前面都是**queue**
      - `if (!td->io_ops->commit)` —— 对于同步操作而言，本身没有commit方法，到此就要结束了，但是为了用异步在**行为上统一**，直接**mark sunmit以及completed**
        - `io_u_mark_submit(td, 1);`
        - `io_u_mark_complete(td, 1);`
      - `if (ret == FIO_Q_COMPLETED)` —— **同步IO完成，同步IO确实会走到这个逻辑来**
      - `else if (ret == FIO_Q_QUEUED)` —— **异步IO有入队**
        - `td->io_u_queued++;`
        - `if (td->io_u_queued >= td->o.iodepth_batch)`
          - `td_io_commit(td);`
            - `if (td->io_ops->commit)`
              - `ret = td->io_ops->commit(td);` —— **以libaio的提交为例**
                - `static int fio_libaio_commit(struct thread_data *td)` —— **异步IO**的**commit方法，同步IO也确实不会有这个方法**
                  - `do {`
                    - `long nr = ld->queued;`
                    - `ret = io_submit(ld->aio_ctx, nr, iocbs);` —— **linux内核的系统调用，即linux原生异步IO的接口**
                    - `fio_libaio_queued(td, io_us, ret);`
                      - `fio_gettime(&now, NULL);`
                      - `for (i = 0; i < nr; i++)`
                        - `struct io_u *io_u = io_us[i];`
                        - `memcpy(&io_u->issue_time, &now, sizeof(now));`
                          - **记录io的issue时间**
                        - `io_u_queued(td, io_u);`
                          - `slat_time = ntime_since(&io_u->start_time, &io_u->issue_time);` —— 异步IO计算 `slat time`
                            - 就是发射时间**减去**提交时间

                            - `add_slat_sample(td, io_u->ddir, slat_time, io_u->xfer_buflen, io_u->offset, io_u->i`
                              —— 这是额外的**统计工作**
                      - `io_u_mark_submit(td, ret);`
                    - `} while (ld->queued);`
    - `io_queue_event(td, io_u, &ret, ddir, &bytes_issued, 0, &comp_time);`
    - `ret = io_u_queued_complete(td, i);`
      - `ret = td_io_getevents(td, min_evts, td->o.iodepth_batch_complete_max, tvp);`
        - `if (max && td->io_ops->getevents)`
          - `r = td->io_ops->getevents(td, min, max, t);` —— **fio_libaio_getevents**
            - `static int fio_libaio_getevents(struct thread_data *td, unsigned int min, unsigned int max, const struct timespec *t)`
              - `do {`
                - `r = io_getevents(ld->aio_ctx, actual_min, max, ld->aio_events + events, lt);`
                  - **io_getevents就是系统调用用了，会阻塞在这里等待内核告诉我IO完成**
              - `} while (events < min);`
      - `init_icd(td, &icd, ret);`
        - `fio_gettime(&icd->time, NULL);` —— **得到一个IO的完成时间 io complete time**
      - `ios_completed(td, &icd);`
      - `for (i = 0; i < icd->nr; i++)`
        - `io_u = td->io_ops->event(td, i);` —— **fio_libaio_event**
        - `io_completed(td, &io_u, icd);`
          - `if (should_account(td))`
            - `account_io_completion(td, io_u, icd, ddir, bytes);`
              - `llnsec = ntime_since(&io_u->issue_time, &icd->time);` —— **完成延迟**
              - `add_clat_sample(td, idx, llnsec, ...)`
              - `tnsec = ntime_since(&io_u->start_time, &icd->time);` —— **总延迟**
              - `add_lat_sample(td, idx, tnsec, ...)`
    - `if (should_fsync(td) && (td->o.end_fsync || td->o.fsync_on_close))`
      - `for_each_file(td, f, i)`
        - `fio_file_fsync(td, f)`
          - `fio_io_sync(td, f);` —— 同步操作可能确实有sync的需要?
  - `update_rusage_stat(td);` —— **详见下文，与CPU时间相关**
  - `td->ts.total_run_time = mtime_since_now(&td->epoch);` —— **fio总时间**

过渡

- fio代码尽管有逻辑上的复杂性，但是基本框架是清晰的，就是**主线程waitpid，然后工作线程执行IO**的框
- 当然fio依然有很多的细节需要探索，下面以快问快答的形式呈现

fio快问快答

1. 工作task到底是多线程还是多进程

- 取决于是否指定 `-thread` 参数，如果指定，则调用 `pthread_create` 创建**工作线程**，否则调用fork创建**子进程**

2. fio是怎么与内核打交道的（这里只关注IO函数）

- 对于同步引擎 `sync/psync` 等等，fio直接调用 `read/write/pread/pwrite` 等库函数，进而直接调用内核提供的读写接口
  - 同步操作， `read/write` 等读写接口均**等待IO结束**后才返回
- 对于异步引擎 `libaio` ，fio最终调用的os接口是 `io_submit` （即linux内核支持的原生异步io）
  - 异步操作， `io_submit` 负责将IO请求丢给内核即返回，随后IO完成后直接通知fio

3. iodepth起什么作用

- fio能够同时提交的IO数量
  - 对于同步IO，iodepth只能是1，提交1个就意味着需要等待完成

        - 所以iodepth**无意义**
    - 对于异步IO，这就是一个有意义的值，它表示fio能够**一次性**丢给OS多少个IO请求

4. fio的请求是否经过buffer cache

- 取决于是否指定 `direct=1` ，direct实际上是open文件时的flag，如果指定了direct=1，则所有的IO请求将bypass掉buffer cache
- bypass掉buffer cache之后的一个重要接口是 `submit_bio`
    - **同步IO**调用 `submit_bio_wait`
    - **异步IO**调用 `submit_bio_nowait` —— 实际内核中这个接口的描述是不准确的，只是为了在表述上**更加清晰**
- 在使用buffer cache的情况下，以写操作为例，实际完成的操作就是 `copy_from_user`

5. fio读写接口的内核IO栈简述

- fio经常测试的文件包括 `/dev/sda` 以及 `/home/zc/workspace/test` 两种文件
    - `/dev/sda` 是块设备文件，代表一块SCSI盘， `/dev/sda` 这个文件本身位于**udev文件系统**
    - 假设 `/` 是**ext4文件系统**的挂载点，则 `/home/zc/workspace/test` 代表**ext4文件系统中**的一个文件
- **write**接口，同步接口
    - write下去先执行的是 `vfs_write` ，然后会执行所在文件系统的**文件的write**方法
        - 例如对于 `/dev/sda` ，写方法就是**blkdev_write_iter**
            - 如果设置了direct，则最终会调用到 `submit_bio_wait`
            - 如果**未**设置direct，则本质就是**copy from user**到内核buffer
        - 对于ext4下的文件，其写方法是**ext4_file_write_iter**
            - 如果未设置direct，则调用**ext4_buffered_write_iter**
                - 依然是 `copy from user` 到内核buffer
            - 如果设置了direct，则调用**ext4_dio_write_iter**
                - 绕了半天还是直接 `submit_bio_wait`
- **io_submit**接口，异步接口
    - 最终调用fop的write方法，基本同上，但是均为no wait方式，**发完命令立刻返回**

6. fio对时间的统计 —— 这个就有点复杂了，细细品味一下

```
# 首先是异步IO
# fio -filename=/dev/sdb -direct=1 -zero_buffers=1 -iodepth=1 -thread -rw=randwrite -ioengine=libaio -bs=4k -size=100G -numjobs=1 -runtime=30 -gr
mytest: (g=0): rw=randwrite, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=libaio, iodepth=1
fio-3.16
Starting 1 thread
Jobs: 1 (f=1): [w(1)][100.0%][w=111MiB/s][w=28.3k IOPS][eta 00m:00s]
mytest: (groupid=0, jobs=1): err= 0: pid=6147: Sun Mar 19 15:31:31 2023
  write: IOPS=28.0k, BW=113MiB/s (119MB/s)(3398MiB/30001msec); 0 zone resets
    slat (nsec): min=1031, max=100349, avg=1684.98, stdev=551.66
    clat (nsec): min=1083, max=2308.8k, avg=32510.78, stdev=11716.18
     lat (usec): min=29, max=2309, avg=34.22, stdev=11.75
    clat percentiles (nsec):
     |  1.00th=[28544],  5.00th=[28544], 10.00th=[28800], 20.00th=[28800],
     | 30.00th=[29056], 40.00th=[29056], 50.00th=[29312], 60.00th=[29568],
     | 70.00th=[29568], 80.00th=[29824], 90.00th=[46848], 95.00th=[61696],
     | 99.00th=[63232], 99.50th=[66048], 99.90th=[69120], 99.95th=[71168],
     | 99.99th=[81408]
   bw (  KiB/s): min=112862, max=120088, per=100.00%, avg=116029.00, stdev=2195.42, samples=59
   iops        : min=28215, max=30022, avg=29007.22, stdev=548.85, samples=59
  lat (usec)   : 2=0.01%, 50=93.67%, 100=6.32%, 250=0.01%, 500=0.01%
  lat (msec)   : 2=0.01%, 4=0.01%
  cpu          : usr=1.83%, sys=7.48%, ctx=869998, majf=0, minf=1
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,869980,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
  WRITE: bw=113MiB/s (119MB/s), 113MiB/s-113MiB/s (119MB/s-119MB/s), io=3398MiB (3563MB), run=30001-30001msec

Disk stats (read/write):
  sdb: ios=52/867102, merge=0/0, ticks=9/28009, in_queue=28017, util=99.77%
###
###
# 再来个同步IO
# fio -filename=/dev/sdb -direct=1 -zero_buffers=1 -iodepth=1 -thread -rw=randwrite -ioengine=psync -bs=4k -size=100G -numjobs=1 -runtime=30 -gro
mytest: (g=0): rw=randwrite, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=psync, iodepth=1
fio-3.16
Starting 1 thread
Jobs: 1 (f=1): [w(1)][100.0%][w=116MiB/s][w=29.6k IOPS][eta 00m:00s]
mytest: (groupid=0, jobs=1): err= 0: pid=6182: Sun Mar 19 15:34:14 2023
  write: IOPS=28.8k, BW=113MiB/s (118MB/s)(3379MiB/30000msec); 0 zone resets
    clat (usec): min=29, max=1432, avg=34.44, stdev=10.03
     lat (usec): min=29, max=1432, avg=34.47, stdev=10.03
    clat percentiles (nsec):
     |  1.00th=[29824],  5.00th=[30080], 10.00th=[30080], 20.00th=[30336],
     | 30.00th=[30592], 40.00th=[30592], 50.00th=[31872], 60.00th=[32128],
     | 70.00th=[32384], 80.00th=[33024], 90.00th=[48384], 95.00th=[63232],
     | 99.00th=[64256], 99.50th=[68096], 99.90th=[70144], 99.95th=[72192],
     | 99.99th=[82432]
   bw (  KiB/s): min=109864, max=118784, per=99.95%, avg=115264.12, stdev=2494.02, samples=59
   iops        : min=27466, max=29696, avg=28816.00, stdev=623.52, samples=59
  lat (usec)   : 50=92.33%, 100=7.66%, 250=0.01%, 1000=0.01%
  lat (msec)   : 2=0.01%
  cpu          : usr=1.29%, sys=4.52%, ctx=864990, majf=0, minf=0
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,864943,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
  WRITE: bw=113MiB/s (118MB/s), 113MiB/s-113MiB/s (118MB/s-118MB/s), io=3379MiB (3543MB), run=30000-30000msec

Disk stats (read/write):
  sdb: ios=52/861918, merge=0/0, ticks=9/28173, in_queue=28182, util=99.79%
```

- 对于IO而言，我们最关注的就是延迟，观察上面的**2个fio结果**
  - 同步IO，**两个**延迟，**clat以及lat**
  - 异步IO，**三个**延迟，**slat/clat以及lat**
- 先看一眼**官方的描述**

    - ref
        - https://fio.readthedocs.io/en/latest/fio_doc.html#interpreting-the-output
            - 到最后还是官方的解释**清晰明了**
    - slat —— Submission latency
        - This is the time from when fio initialized the I/O to submission —— 就是字面意思，**IO**从**初始化**到**提交**的时间
            - For synchronous ioengines this includes the time up until just before the ioengine's queue function is called
                - For sync I/O this row is not displayed as the slat is **negligible**
                    - 对于同步引擎，**准备到入队**（queue）几乎可以**忽略不计**，所以压根不用显示
            - For asynchronous ioengines this includes the time up through the completion of the ioengine's queue function (and commit function if it is defined)
                - 对于异步IO，可以通俗的理解为**准备时间**，就是**准备IO并且入队，最后再提交**的这段时间
                - 其实就是异步IO才有**提交延迟**这个概念
    - clat —— Completion latency
        - Same names as slat, this denotes the time from submission to completion of the I/O pieces —— **字面意思就是提交到完成的时间**
            - For sync I/O, this represents the time from when the I/O was submitted to the operating system to when it was completed
                - 对于同步IO，这个时间就是IO提交到内核直到它完成的时间
            - For asynchronous ioengines this is the time from when the ioengine's queue (and commit if available) functions were completed to when the I/O's completion was reaped by fio
                - **io_submit函数完成的时间，其实也是内核里IO的完成时间**
    - lat —— Total latency —— 总延迟就是两段**加起来**
        - Same names as slat and clat, this denotes the time from when fio created the I/O unit to completion of the I/O operation
        - It is the sum of submission and completion latency
- **再看一眼这个具体是怎么算的**
    - 首先是 `io_unit` 的时间计算
        - `static void do_io(struct thread_data *td, uint64_t *bytes_done)` —— **一个单独的IO task**
            - `total_bytes = td->o.size;`
            - `io_u = get_io_u(td);` —— `io_u` 是追踪每一个IO的handle，fio为**每一个IO**维护一个 `io_unit`
                - `io_u = __get_io_u(td);`
                - `if (!td->o.disable_lat)`
                    - `fio_gettime(&io_u->start_time, NULL);` —— **记录io的start_time，每一个IO都会记录**
            - `ret = io_u_submit(td, io_u);`
                - `return td_io_queue(td, io_u);`
                    - `queue/commit` —— 提交的操作**异步IO好理解**
                    - `fio_gettime(&io_u->issue_time, NULL);` —— **记录io的issue_time，每一个IO都会记录**
            - `ret = io_u_queued_complete(td, i);`
                - `init_icd(td, &icd, ret);`
                    - `fio_gettime(&icd->time, NULL);` —— **得到一个IO的完成时间 io complete time**
                - `ios_completed(td, &icd);`
    - 所以对于一个 `io_unit` （**一个IO**，其实可以理解为一次 `read/write/io_submit..` 系统调用）来说，关键的3个时间点包括
        1. `fio_gettime(&io_u->start_time, NULL);` —— **IO的开始时间**
        2. `fio_gettime(&io_u->issue_time, NULL);` —— **IO的发射时间**
        3. `fio_gettime(&icd->time, NULL);` —— **IO的完成时间**
    - `slat/clat/lat` 的计算
        - **slat**
            - `slat_time = ntime_since(&io_u->start_time, &io_u->issue_time);`
                - **IO的发射时间减去开始时间**
            - `add_slat_sample(td, io_u->ddir, slat_time, ...);` —— **统计**
        - **clat**
            - `llnsec = ntime_since(&io_u->issue_time, &icd->time);` —— **完成延迟**
            - `add_clat_sample(td, idx, llnsec, ...)` —— **统计**
        - **lat**
            - `tnsec = ntime_since(&io_u->start_time, &icd->time);` —— **总延迟**
            - `add_lat_sample(td, idx, tnsec, ...)` —— **统计**
- 看一眼add_stat_sample
    - `add_slat_sample` / `add_clat_sample` / `add_lat_sample` --**记录提交延迟/完成延迟/总延迟**
        - `add_stat_sample`

```
static inline void add_stat_sample(struct io_stat *is, unsigned long long data)
{
    double val = data; /* data是本次io的延迟 */
    double delta;

    if (data > is->max_val)
        is->max_val = data; /* 记录最大值 */
    if (data < is->min_val)
        is->min_val = data; /* 记录最小值 */

    /*
        假设 a(1) .... a(n) 的mean = sum(n)/n
        delta = a(n+1) - mean(n)
        mean(n+1) =  mean(n) + delta/(n+1)
                = sum(n)/n + delta/(n+1)
                ...
                = sum(n+1)/(n+1)
    */
    delta = val - is->mean.u.f;
    if (delta) {
        is->mean.u.f += delta / (is->samples + 1.0);
        is->S.u.f += delta * (val - is->mean.u.f);
    }

    is->samples++;
}
```

7. cpu时间

```
# 首先是异步IO
# fio -filename=/dev/sdb -direct=1 -zero_buffers=1 -iodepth=1 -thread -rw=randwrite -ioengine=libaio -bs=4k -size=100G -numjobs=1 -runtime=30 -gr
cpu          : usr=1.83%, sys=7.48%, ctx=869998, majf=0, minf=1
###
###
# 再来个同步IO
# fio -filename=/dev/sdb -direct=1 -zero_buffers=1 -iodepth=1 -thread -rw=randwrite -ioengine=psync -bs=4k -size=100G -numjobs=1 -runtime=30 -gro
cpu          : usr=1.29%, sys=4.52%, ctx=864990, majf=0, minf=0
```

- 看一下CPU时间是什么
  - cpu —— CPU usage
    - User and system time, along with the number of context switches this thread went through, usage of system and user time, and finally the number of major and minor page faults
    - The CPU utilization numbers are averages for the jobs in that reporting group, while the context and fault counters are summed
- 重点看一下CPU时间是如何统计出来的，单个work线程为例，代码调用如下

```
/* This describes a single thread/process executing a fio job. */
struct thread_data {
    /* ... */
        struct rusage ru_start;
        struct rusage ru_end;
    /* ... */
}
/* /usr/include/x86_64-linux-gnu/bits/types/struct_rusage.h */
/* 系统中存在这个数据结构 */
struct rusage
{
    /* Total amount of user time used.  */
    struct timeval ru_utime;
    /* Total amount of system time used.  */
    struct timeval ru_stime;
    /* ... */
}
```

- 首先单独介绍一个函数**GetProcessTimes()**

```
int getrusage(int who, struct rusage *r_usage)
{
    /* 进程创建和退出时间是时间点，表示为自1601年1月1日，1601年1月1日午夜以来在英格兰格林威治运行的时间量。应用程序可以使用多个函数将此类值转换为更通用的表单
        const uint64_t SECONDS_BETWEEN_1601_AND_1970 = 11644473600;
    /* 所有时间都使用 FILETIME 数据结构表示。 这种结构包含两个 32 位值，这些值组合成 64 位计数 100 纳秒的时间单位 */
        FILETIME cTime; /* create time，指向接收进程创建时间的 FILETIME 结构的指针 */
    FILETIME eTime; /* exit time，指向接收进程退出时间的 FILETIME 结构的指针。如果进程尚未退出，则此结构的内容未定义 */
    FILETIME kTime; /* kernel time，该结构接收进程在内核模式下执行的时间量 */
    FILETIME uTime; /* usr time，该结构接收进程在用户模式下执行的时间量 */
        time_t time;
        HANDLE h;

        memset(r_usage, 0, sizeof(*r_usage));

    h = GetCurrentProcess(); /* 只是获得当前的进程handle，表示我的目标是该进程的cpu usage */
    GetProcessTimes(h, &cTime, &eTime, &kTime, &uTime);

        time = ((uint64_t)uTime.dwHighDateTime << 32) + uTime.dwLowDateTime;
        /* Divide by 10,000,000 to get the number of seconds and move the epoch from
         * 1601 to 1970 */
        time = (time_t)(((time)/10000000) - SECONDS_BETWEEN_1601_AND_1970);
    /////////////////////
        r_usage->ru_utime.tv_sec = time; /* Total amount of user time used */
        /////////////////////
    /* getrusage() doesn't care about anything other than seconds, so set tv_usec to 0 */
        r_usage->ru_utime.tv_usec = 0;
        time = ((uint64_t)kTime.dwHighDateTime << 32) + kTime.dwLowDateTime;
        /* Divide by 10,000,000 to get the number of seconds and move the epoch from
         * 1601 to 1970 */
        time = (time_t)(((time)/10000000) - SECONDS_BETWEEN_1601_AND_1970);
        /////////////////////
    r_usage->ru_stime.tv_sec = time; /* Total amount of system time used */
        r_usage->ru_stime.tv_usec = 0;
    /////////////////////
        return 0;
}
```

- `thread_main`
    - `set_epoch_time(td, o->log_unix_epoch | o->log_alternate_epoch, o->log_alternate_epoch_clock_id);` —— **总的开始时间**
    - `fio_getrusage(&td->ru_start);`
        - `return getrusage(RUSAGE_SELF, ru);` —— **统计CPU利用率相关**
            - **为ru_start赋值**
    - `do_io(td, bytes_done);` —— **真正执行IO的时间**
    - `update_rusage_stat(td);` —— **统计CPU利用率相关**
        - `fio_getrusage(&td->ru_end);`
            - `return getrusage(RUSAGE_SELF, ru);`
                - **为ru_end赋值**
        - `ts->usr_time += mtime_since_tv(&td->ru_start.ru_utime, &td->ru_end.ru_utime);`
            - **说白了就是end-start**
        - `ts->sys_time += mtime_since_tv(&td->ru_start.ru_stime, &td->ru_end.ru_stime);`
            - **说白了就是end-start**
        - `ts->ctx += td->ru_end.ru_nvcsw + td->ru_end.ru_nivcsw - (td->ru_start.ru_nvcsw + td->ru_start.ru_nivcsw);`
        - `ts->minf += td->ru_end.ru_minflt - td->ru_start.ru_minflt;`
        - `ts->majf += td->ru_end.ru_majflt - td->ru_start.ru_majflt;`
        - `memcpy(&td->ru_start, &td->ru_end, sizeof(td->ru_end));`
    - `td->ts.total_run_time = mtime_since_now(&td->epoch);` —— **fio总时间**

所以关键还是函数GetProcessTimes()

- **TODO** —— 为什么它可以在linux上跑

8. fio关于CPU亲和性的设置

- `cpus_allowed=str`

  - Controls the **same options as cpumask**, but accepts a textual specification of the permitted CPUs instead and CPUs are indexed from 0
    - So to use CPUs 0 and 5 you would specify `cpus_allowed=0,5`
    - This option also allows a range of CPUs to be specified – say you wanted a binding to CPUs 0, 5, and 8 to 15
      - you would set `cpus_allowed=0,5,8-1`
- `cpus_allowed_policy=str`
  - Two policies are supported
    - **shared**
      - All jobs will share the CPU set specified
    - **split**
      - Each job will get a unique CPU from the CPU set
  - **shared** is the default behavior, if the option isn’t specified. If split is specified, then fio will assign **one cpu per job**. If not enough CPUs are given for the jobs listed, then fio will **roundrobin** the CPUs in the set
    - **CPU小于jobs数量的话就roundrobin**
- `cpumask` —— Set the CPU affinity of this job
  - The parameter given is a bit mask of allowed CPUs the job may run on
  - So if you want the allowed CPUs to be 1 and 5, you would pass the decimal value of `(1 << 1 | 1 << 5)` or `34`
- `numa_cpu_nodes=str`
  - **指定numa node**
- `...`

9. 时钟校准细节

10. fio对架构的依赖以及库的依赖以及多平台的支持 —— TODO

- Fio works on (at least) **Linux**, Solaris, AIX, HP-UX, OSX, NetBSD, OpenBSD, **Windows**, FreeBSD, and DragonFly

11. fio之merge —— 测试fio几个点（顺序小写/修改块设备的调度方式...）

```
mytest: (g=0): rw=randwrite, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=libaio, iodepth=1
fio-3.16
Starting 1 thread
Jobs: 1 (f=1): [w(1)][100.0%][w=111MiB/s][w=28.3k IOPS][eta 00m:00s]
mytest: (groupid=0, jobs=1): err= 0: pid=6147: Sun Mar 19 15:31:31 2023
  write: IOPS=28.0k, BW=113MiB/s (119MB/s)(3398MiB/30001msec); 0 zone resets
    slat (nsec): min=1031, max=100349, avg=1684.98, stdev=551.66
    clat (nsec): min=1083, max=2308.8k, avg=32510.78, stdev=11716.18
     lat (usec): min=29, max=2309, avg=34.22, stdev=11.75
    clat percentiles (nsec):
     |  1.00th=[28544],  5.00th=[28544], 10.00th=[28800], 20.00th=[28800],
     | 30.00th=[29056], 40.00th=[29056], 50.00th=[29312], 60.00th=[29568],
     | 70.00th=[29568], 80.00th=[29824], 90.00th=[46848], 95.00th=[61696],
     | 99.00th=[63232], 99.50th=[66048], 99.90th=[69120], 99.95th=[71168],
     | 99.99th=[81408]
   bw (  KiB/s): min=112862, max=120088, per=100.00%, avg=116029.00, stdev=2195.42, samples=59
   iops        : min=28215, max=30022, avg=29007.22, stdev=548.85, samples=59
  lat (usec)   : 2=0.01%, 50=93.67%, 100=6.32%, 250=0.01%, 500=0.01%
  lat (msec)   : 2=0.01%, 4=0.01%
  cpu          : usr=1.83%, sys=7.48%, ctx=869998, majf=0, minf=1
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,869980,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
  WRITE: bw=113MiB/s (119MB/s), 113MiB/s-113MiB/s (119MB/s-119MB/s), io=3398MiB (3563MB), run=30001-30001msec

Disk stats (read/write):
  sdb: ios=52/867102, merge=0/0, ticks=9/28009, in_queue=28017, util=99.77%
```

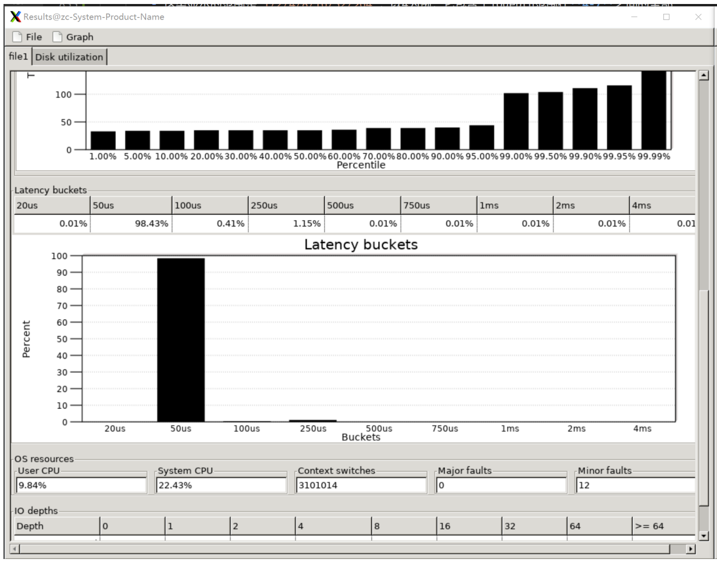
- 先就这这个结果将之前没有太留意过的性能指标看一看 —— https://fio.readthedocs.io/en/latest/fio_doc.html#interpreting-the-output

    - **issued rwts**表示发出的IO次数等，最后一行**Disk stats**也表示的IO状态，区别在于**issued rwts**是fio自己准确统计出来的，而**Disk stats**来自于 `/sys/block/nvme0n1/stat` ，后面的值是有定时器任务在读取的，结合白柯凡对fio源码的阅读，这里的统计值不一定是准确的，所以最终io次数以rwts为准，**rwts含义如下**
        - **rwts: read/write/trim/sync**
            - **read/write/trim** —— The number of read/write/trim requests issued, and how many of them were short or dropped
                - **就是这些request的发出次数**
            - **sync:fsync=int** —— 对于非**direct=1**的操作有意义，**每多少个IO sync一次**
- 然后从**IO depths**开始有一系列的参数简单描述一下
    - **IO depths** —— io depth的分布情况，假设iodepth=2，那么在fio本次测试的整个生命周期中iodepth大部分将是2，极少量为1
        - 这里显示的范围是 `1/2/4/8/16/32/>64` ，以4为例，它包含了iodepth范围在**4-7**之间的全部
    - **IO submit**
        - How many pieces of I/O were submitting in a single submit call
            - 最开始让我好生疑惑的说，为啥我测得结果都是**4**，不应该都是**1**么
            - 后来仔细看了一下，确实应该都是1，因为4 cover的是**1-4**的全部可能
    - **IO complete**
        - 同上
    - **IO issued rwts** —— 详见上文
    - **IO latency** —— 看起来是个相对高级的功能，先略过
- stats统计的状态值 —— `Documentation/block/stat.rst` —— 都是**前读后写**
    - merge —— fio能够看到merge的
        - 首先fio的direct仅仅是bypass page cache，只是直接到了块层，块层的调度以及电梯什么一个都**不会拉下**，所以还是需要分析一下
        - **Number of merges performed by the I/O scheduler**
            - merge的含义是非常**简单粗暴的**
    - ios —— number of read/write I/Os processed
    - merge —— number of read/write I/0s merged with in-queue I/0
    - ticks —— milliseconds —— total wait time for read/write reguests
    - in_queue —— milliseconds —— total wait time for all requestsmilliseconds
    - util —— TODO **交给俊伟与柯凡一起吧**

再来看一下iops测试异常的一些情况

```
# 这是512字节的随机小写
zc@zc-System-Product-Name:~$ sudo fio -filename=/dev/sdb -direct=1 -iodepth=4 -thread -rw=randwrite -ioengine=libaio -bs=512 -size=100G -numjobs=
mytest: (g=0): rw=randwrite, bs=(R) 512B-512B, (W) 512B-512B, (T) 512B-512B, ioengine=libaio, iodepth=4
fio-3.16
Starting 1 thread
Jobs: 1 (f=1): [w(1)][100.0%][w=39.5MiB/s][w=80.9k IOPS][eta 00m:00s]
mytest: (groupid=0, jobs=1): err= 0: pid=24228: Mon Aug 28 10:56:37 2023
  write: IOPS=80.3k, BW=39.2MiB/s (41.1MB/s)(1177MiB/30001msec); 0 zone resets
    slat (nsec): min=623, max=243070, avg=2883.29, stdev=1120.31
    clat (usec): min=10, max=3754, avg=46.39, stdev=20.36
     lat (usec): min=17, max=3756, avg=49.33, stdev=20.21
    clat percentiles (usec):
     |  1.00th=[   29],  5.00th=[   31], 10.00th=[   33], 20.00th=[   35],
     | 30.00th=[   36], 40.00th=[   39], 50.00th=[   43], 60.00th=[   48],
     | 70.00th=[   51], 80.00th=[   56], 90.00th=[   62], 95.00th=[   78],
     | 99.00th=[  111], 99.50th=[  121], 99.90th=[  167], 99.95th=[  186],
     | 99.99th=[  322]
   bw (  KiB/s): min=39106, max=40580, per=99.99%, avg=40161.31, stdev=276.76, samples=59
   iops        : min=78212, max=81160, avg=80322.63, stdev=553.50, samples=59
  lat (usec)   : 20=0.01%, 50=67.56%, 100=31.01%, 250=1.41%, 500=0.02%
  lat (usec)   : 750=0.01%, 1000=0.01%
  lat (msec)   : 2=0.01%, 4=0.01%
  cpu          : usr=8.99%, sys=30.79%, ctx=807038, majf=0, minf=1
  IO depths    : 1=0.1%, 2=0.1%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,2410106,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=4

Run status group 0 (all jobs):
  WRITE: bw=39.2MiB/s (41.1MB/s), 39.2MiB/s-39.2MiB/s (41.1MB/s-41.1MB/s), io=1177MiB (1234MB), run=30001-30001msec

Disk stats (read/write):
  sdb: ios=86/2401510, merge=0/0, ticks=15/105299, in_queue=105314, util=99.79%

# 这是4K的随机小写
zc@zc-System-Product-Name:~$ sudo fio -filename=/dev/sdb -direct=1 -iodepth=4 -thread -rw=randwrite -ioengine=libaio -bs=4k -size=100G -numjobs=1
mytest: (g=0): rw=randwrite, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=libaio, iodepth=4
fio-3.16
Starting 1 thread
Jobs: 1 (f=1): [w(1)][100.0%][w=405MiB/s][w=104k IOPS][eta 00m:00s]
mytest: (groupid=0, jobs=1): err= 0: pid=24304: Mon Aug 28 10:57:52 2023
  write: IOPS=104k, BW=408MiB/s (428MB/s)(11.0GiB/30001msec); 0 zone resets
    slat (nsec): min=659, max=224620, avg=1251.06, stdev=443.76
    clat (usec): min=19, max=3044, avg=36.77, stdev=11.68
     lat (usec): min=26, max=3046, avg=38.05, stdev=11.69
    clat percentiles (usec):
     |  1.00th=[   33],  5.00th=[   34], 10.00th=[   34], 20.00th=[   35],
     | 30.00th=[   35], 40.00th=[   35], 50.00th=[   35], 60.00th=[   36],
     | 70.00th=[   39], 80.00th=[   39], 90.00th=[   40], 95.00th=[   45],
     | 99.00th=[   68], 99.50th=[   92], 99.90th=[  108], 99.95th=[  125],
     | 99.99th=[  293]
   bw (  KiB/s): min=408544, max=424888, per=100.00%, avg=417787.58, stdev=4487.60, samples=59
   iops        : min=102136, max=106222, avg=104446.88, stdev=1121.90, samples=59
  lat (usec)   : 20=0.01%, 50=98.23%, 100=1.32%, 250=0.43%, 500=0.01%
  lat (usec)   : 750=0.01%, 1000=0.01%
  lat (msec)   : 2=0.01%, 4=0.01%
  cpu          : usr=9.44%, sys=22.56%, ctx=3118306, majf=0, minf=1
  IO depths    : 1=0.1%, 2=0.1%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,3133242,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=4

Run status group 0 (all jobs):
  WRITE: bw=408MiB/s (428MB/s), 408MiB/s-408MiB/s (428MB/s-428MB/s), io=11.0GiB (12.8GB), run=30001-30001msec

Disk stats (read/write):
  sdb: ios=86/3122615, merge=0/0, ticks=15/113417, in_queue=113433, util=99.80%
```

- 其实这个问题最早发现来源于创哥，创哥说要给我看一眼，后来一看确实值得去仔细分析一下



Latency buckets

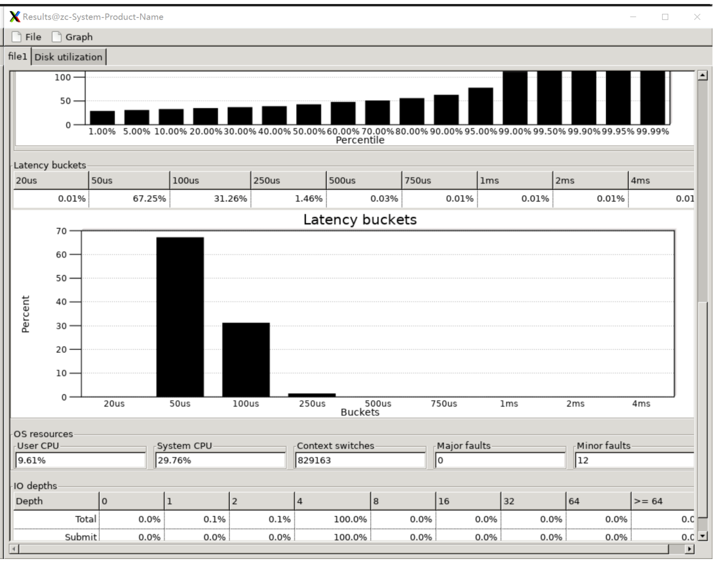
20us	50us	100us	250us	500us	750us	1ms	2ms	4ms
0.01%	98.43%	0.41%	1.15%	0.01%	0.01%	0.01%	0.01%	0.01

OS resources

User CPU: 9.84%
System CPU: 22.43%
Context switches: 3101014
Major faults: 0
Minor faults: 12

IO depths

Depth	0	1	2	4	8	16	32	64	>= 64



Latency buckets

20us	50us	100us	250us	500us	750us	1ms	2ms	4ms
0.01%	67.25%	31.26%	1.46%	0.03%	0.01%	0.01%	0.01%	0.01

OS resources

User CPU: 9.61%
System CPU: 29.76%
Context switches: 829163
Major faults: 0
Minor faults: 12

IO depths

Depth	0	1	2	4	8	16	32	64	>= 64
Total	0.0%	0.1%	0.1%	100.0%	0.0%	0.0%	0.0%	0.0%	0.0
Submit	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	0.0

- **随机写，io粒度越小，iops越小**
  - `fio -filename=/dev/sda -direct=1 -iodepth=4 -thread -rw=randwrite -ioengine=libaio -bs=4k -size=100G -numjobs=1 -runtime=30 -group_reportir`
    - 表现
      - 将iodepth的深度设定为4，bs分别为4k与512，512的iops显著降低，通过分析可知是当IO粒度为512字节时，尾延迟很明显，如上图所示
  - 其实嘉正给我的结果表示这个尾延迟是硬件造成的，**我需要验证一下排除软件尾延迟的嫌疑**，其实用炜杰的钩子就能确定这一点，那个统计的绝对是**驱动+硬件的延迟**
    - 我还需要把dmesg的输出重定向到文件中，然后去check尾延迟是否是硬件造成的，确实是硬件造成的，**但是这能导致iops如此剧烈的降低么**，分析一下上面2个测试的直观结果
      - 测试时间30s，bs=512时发出的IO数量为**2410106**，bs=4k时发出的IO数量为**3133242**，很明显bs=512时所发出的IO数量明显少于bs=4k时所发出的IO数量
        - 当然说白了这就是IOPS最直接的反应
      - 在bs=512时的尾延迟明显大于bs=4k时的尾延迟，且平均延迟更大，延迟方差更明显。延迟大，单位时间能够发出的IO数量自然就少，**所有的锅就甩给硬件的尾延迟效应**

```
# 4k的顺序写
zc@zc-System-Product-Name:~/workspace/fio_example$ sudo fio -filename=/dev/sdb -direct=1 -iodepth=4 -thread -rw=write -ioengine=libaio -bs=4k -si
[sudo] password for zc:
mytest: (g=0): rw=write, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=libaio, iodepth=4
fio-3.16
Starting 1 thread
Jobs: 1 (f=1): [W(1)][100.0%][w=411MiB/s][w=105k IOPS][eta 00m:00s]
mytest: (groupid=0, jobs=1): err= 0: pid=24484: Mon Aug 28 11:13:07 2023
  write: IOPS=106k, BW=415MiB/s (435MB/s)(12.2GiB/30001msec); 0 zone resets
    slat (nsec): min=649, max=227975, avg=1197.36, stdev=332.69
    clat (usec): min=22, max=2520, avg=36.22, stdev=13.02
     lat (usec): min=26, max=2521, avg=37.44, stdev=13.03
    clat percentiles (usec):
     |  1.00th=[   34],  5.00th=[   34], 10.00th=[   35], 20.00th=[   35],
     | 30.00th=[   35], 40.00th=[   35], 50.00th=[   35], 60.00th=[   36],
     | 70.00th=[   36], 80.00th=[   37], 90.00th=[   39], 95.00th=[   43],
     | 99.00th=[   99], 99.50th=[  101], 99.90th=[  106], 99.95th=[  109],
     | 99.99th=[  269]
   bw (  KiB/s): min=415544, max=427744, per=99.99%, avg=424772.49, stdev=2953.53, samples=59
   iops        : min=103886, max=106936, avg=106193.10, stdev=738.38, samples=59
  lat (usec)   : 50=98.73%, 100=0.55%, 250=0.71%, 500=0.01%, 750=0.01%
  lat (usec)   : 1000=0.01%
  lat (msec)   : 2=0.01%, 4=0.01%
  cpu          : usr=8.52%, sys=22.55%, ctx=3179390, majf=0, minf=1
  IO depths    : 1=0.1%, 2=0.1%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,3186316,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=4

Run status group 0 (all jobs):
  WRITE: bw=415MiB/s (435MB/s), 415MiB/s-415MiB/s (435MB/s-435MB/s), io=12.2GiB (13.1GB), run=30001-30001msec

Disk stats (read/write):
  sdb: ios=86/3175530, merge=0/0, ticks=15/113650, in_queue=113664, util=99.81%

# 512字节的顺序写
zc@zc-System-Product-Name:~/workspace/fio_example$ sudo fio -filename=/dev/sdb -direct=1 -iodepth=4 -thread -rw=write -ioengine=libaio -bs=512 -s
mytest: (g=0): rw=write, bs=(R) 512B-512B, (W) 512B-512B, (T) 512B-512B, ioengine=libaio, iodepth=4
fio-3.16
Starting 1 thread
Jobs: 1 (f=1): [W(1)][100.0%][w=79.7MiB/s][w=163k IOPS][eta 00m:00s]
mytest: (groupid=0, jobs=1): err= 0: pid=24551: Mon Aug 28 11:13:45 2023
  write: IOPS=162k, BW=79.3MiB/s (83.1MB/s)(2378MiB/30001msec); 0 zone resets
    slat (nsec): min=612, max=38491, avg=888.44, stdev=236.30
    clat (usec): min=7, max=840, avg=23.57, stdev= 5.30
     lat (usec): min=16, max=841, avg=24.48, stdev= 5.31
    clat percentiles (usec):
     |  1.00th=[   19],  5.00th=[   19], 10.00th=[   21], 20.00th=[   22],
     | 30.00th=[   23], 40.00th=[   23], 50.00th=[   23], 60.00th=[   24],
     | 70.00th=[   24], 80.00th=[   26], 90.00th=[   28], 95.00th=[   30],
     | 99.00th=[   37], 99.50th=[   38], 99.90th=[  102], 99.95th=[  105],
     | 99.99th=[  111]
   bw (  KiB/s): min=79896, max=82012, per=99.98%, avg=81151.17, stdev=540.57, samples=59
   iops        : min=159792, max=164024, avg=162302.41, stdev=1081.11, samples=59
  lat (usec)   : 10=0.01%, 20=9.39%, 50=90.41%, 100=0.07%, 250=0.12%
  lat (usec)   : 500=0.01%, 750=0.01%, 1000=0.01%
  cpu          : usr=8.80%, sys=24.22%, ctx=3416339, majf=0, minf=1
  IO depths    : 1=0.1%, 2=0.1%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,4870258,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=4

Run status group 0 (all jobs):
  WRITE: bw=79.3MiB/s (83.1MB/s), 79.3MiB/s-79.3MiB/s (83.1MB/s-83.1MB/s), io=2378MiB (2494MB), run=30001-30001msec

Disk stats (read/write):
  sdb: ios=49/4853626, merge=0/0, ticks=8/110510, in_queue=110519, util=99.78%
```

- **顺序写，io粒度越小，iops越大**
  - `fio -filename=/dev/sda -direct=1 -iodepth=4 -thread -rw=write -ioengine=libaio -bs=4k -size=100G -numjobs=1 -runtime=30 -group_reporting -r`
    - 表现
      - **4k的顺序写iops=105K，512的顺序写iops=163k，merge均为0**

        - TODO —— **找柯凡和俊伟**
- 再说回调度器，先说结论，**调度器没啥关系**
    - `/sys/block/sda/queue/scheduler` 的取值可能是 `elevator_name/none/deadline` 这其实就是电梯的名字
    - 但是根据之前所看到的结果，电梯只占了一小部分，还有其它各种机制，例如 `plug_list` ，所以电梯真的会有如此大的影响么
        - **确实与调度没什么吊关系**

12. fio之异步IO —— 队列深度加倍（1-2）iops变化情况的问题讨论 —— 这里其实延伸了很多问题出来，都解释一下

- 先说一些有意思的发现
    - 我自己nvme ssd实验盘的iops不稳定。最开始发现我的盘在 `iodepth=1` 的时候，iops约为70k左右，但是 `iodepth=2` 的时候，iops约为160k左右，显然这个iops不是2倍的增长，进而引发了后续的问题。然后找别人测试的时候发现别人的nvme ssd在iodepth由1到2的过程中iops的增长还是比较符合2倍关系的。直到我用上了gfio图形化动态界面的时候我发现，淦，我自己nvme ssd的这个iops性能极其不稳定，偶尔也能稳定在80多k的样子，但是有时候iops基本都稳定在了70k左右
    - gfio的使用也是一个有趣的发现，这玩意真好用，能让我看到动态的变化关系
    - 然后还发现**9400-16i**配合联想盘，当**iodepth=1**的时候iops只有**7-8k**，当**iodepth=2**的时候，iops相对正常达到**44k**
        - **iodepth=1**时的iops显然不符合盘的性能，算是一个有趣的发现吧，这个时候单个io的平均延迟在**129us**左右，而**iodepth=2**时的平均延迟在**42us**左右
        - **9300-8i**配联想400G小盘，在**iodepth=1**的时候，iops约为**30k**左右，平均延迟为**31us**，当**iodepth=2**的时候，iops可达到**64k**
        - 但是dell的盘在**93/94**上相对都正常，**iodepth由1到2**的过程，iops典型的由**30k到60k**

gfio

- 首先安装gfio
    - `sudo apt-get install libcanberra-gtk-module`
    - `sudo apt-get install gfio`
- 新建一个终端，以sudo身份启动fio
    - `sudo fio -S`
- 在另一个终端中运行**gfio**，该终端（**例如mobaxterm**）一定要有图形界面服务

回顾一下之前炜杰搞的那个钩子

- 之前一直觉得内核提供的钩子不够用，但是后来仔细思考了一下，这个足够用了
    - 因为命令发射的位置trace的函数是 `rtn = host->hostt->queuecommand(host, cmd);`
        - `trace_scsi_dispatch_cmd_start(cmd);`
        - `rtn = host->hostt->queuecommand(host, cmd);`
    - 一般来说，queuecommand是驱动提供的，在该函数中做了什么驱动一清二楚，基本不会有性能损耗的操作，而且上述位置也是公共位置的最后一个位置，该钩子统计的结果与fio的统计结果基本**无差**，实际上queuecommand中只做两件事
        - **if not return**
        - **issue**
- 来总结一下炜杰这个的基本实现原理
    - 首先根据 `find_kallsyms_lookup_name` ，通过字符串找函数名
    - 然后通过找到 `__tracepoint_scsi_dispatch_cmd_start` ，确定这里的一个回调函数，注册一个钩子函数，在开始与结束的时候做一些逻辑，当然这些逻辑本身也会影响准确性，需要尽量的降低这些影响
    - 一个cmd 开始的钩子函数是 `_scsi_dispatch_cmd_start`
    - 一个cmd 结束的钩子函数是 `_scsi_dispatch_cmd_done`

来整一下iops的计算公式

- 需要有一个简单的模型来计算iops，相关假设如下
    - 软件发射io到io进入硬件的时间记为**Tissue**
        - 完全由软件决定，可以非常小，**甚至就是一条指令的差别**
    - io进入硬件队列后，在队列上的移动时间记为**Tmove**
        - 请求在队列上移动的时间，**也理应非常小**
    - io进入硬件到达队首后，到io处理完成的时间记为**Tdelay**
        - **这就是真正IO的时间**
- 所以一个io从发射到完成的时间记为**Tissue+Tmove+Tdelay**，当**iodepth=1**时，**Tmove=0**。所以最终iops的计算公式如下图，**目前均不考虑iodepth超过硬件限制的情况**

    - **iodepth=1**
        - $iops = \frac{1}{\frac{Tissue+Tdelay}{1}}$
    - **iodepth=2**
        - $iops = \frac{1}{\frac{Tissue+Tdelay+Tissue+Tmove}{2}}$
    - **iodepth=3**
        - $iops = \frac{1}{\frac{Tissue+Tdelay+Tissue+Tmove+Tissue+2*Tmove}{3}}$
    - **iodepth=n**
        - $iops = \frac{1}{\frac{n*Tissue+Tdelay+Tmove+2*Tmove+...+(n-1)*Tmove}{n}} = \frac{1}{\frac{n*Tissue+Tdelay+(\frac{n(n-1)}{2})*Tmove}{n}} = \frac{n}{n*Tissue+Tdelay+(\frac{n(n-1)}{2})*Tmove}$
- 分析一下iops的计算公式
    - 先求导看一下单调性，细节不看了，直接上结论
        - 取决于Tdelay与Tmove的关系，超过一定范围，iops由单调递增变为单调递减，但是考虑Tmove非常小，所以这个n的取值要非常大，**iops才可能降低**
    - 整个iops与什么有关呢，主要还是取决于硬件的两个指标， `Tdelay` 与 `Tmove`
- 当开始考虑**iodepth超过硬件的队列深度**的情况时，需要额外考虑一些内容
    - 考虑流水线的知识，iops会有所降低

blktrace blkparse btt

- 这个work交给柯凡搞过，他找到了一些统计IO延迟的工具，如上所示
    - 有用到的时候回来再看看的说

13. 其它可配置参数 —— 持续更新中

- `--idle-prof=percpu`
    - As system but also show per CPU idleness
- Verification —— 实际使用过程中需要 `... -do_verify=1 -verify=crc32 ...`
    - `do_verify=1`
        - Run the verify phase after a write phase. Only valid if verify is set. Default: true
            - 这玩意默认是true的哦吼，那其实只需要指定一个verify的pattern就可以了
    - `verify=md5` —— 方式是多样的 —— `crc64/crc32c...`
        - If writing to a file, fio can verify the file contents after each iteration of the job
    - 带校验的写确实是先写一次，再读一次；带校验的时候，**zero_buffer**没啥用
- `zero_buffers`
    - Initialize buffers with **all zeros**
    - Default: fill buffers with **random data**
- fio不指定offset则偏移为0