STORAGE DEVELOPER CONFERENCE
SDC 23
FREMONT MARRIOTT SILICON VALLEY
SEPTEMBER 18-21, 2023
BY Developers FOR Developers

io_uring

Status Update within Samba

Stefan Metzmacher <metze@samba.org>

Samba Team / SerNet

2023-09-20

https://samba.org/~metze/presentations/2023/SDC/

Check for Updates

- Check for an updated version of this presentation here:
- https://samba.org/~metze/presentations/2023/SDC/

Topics

- What is io-uring?
- io-uring for Samba
- Performance research, prototyping and ideas
- The road to upstream
- Future Improvements
- Questions? Feedback!

Last Status Updates (SDC 2020/2021 - SambaXP 2023)

- I gave a similar talk at the storage developer conference 2020:
  - See https://samba.org/~metze/presentations/2020/SDC/
  - It explains the milestones and design up to Samba 4.13 (in detail)
- I gave a similar talk at the storage developer conference 2021:
  - See https://samba.org/~metze/presentations/2021/SDC/
  - It explains the milestones and updates up to Samba 4.15 (in detail)
- I gave a similar talk at the SambaXP conference 2023:
  - See https://samba.org/~metze/presentations/2023/SambaXP/
  - It explains the milestones and updates up to Samba 4.19 (in detail)

What is io-uring? (Part 1)

- Linux 5.1 introduced a new scalable AIO infrastructure
  - It's designed to avoid syscalls as much as possible
  - kernel and userspace share mmap'ed rings:
    - submission queue (SQ) ring buffer
    - completion queue (CQ) ring buffer
  - See "Ringing in a new asynchronous I/O API" on LWN.NET
- This can be nicely integrated with our async tevent model
  - It may delegate work to kernel threads
  - It seems to perform better compared to our userspace threadpool
  - It can also inline non-blocking operations

io-uring for Samba (Part 1)

- Between userspace and filesystem (available from 5.1):
  - IORING_OP_READV, IORING_OP_WRITEV and IORING_OP_FSYNC
  - Supports buffered and direct io
  - IORING_OP_FSETXATTR, IORING_OP_FGETXATTR (from 5.19)
  - IORING_OP_GETDENTS, under discussion, but seems to be tricky
  - IORING_OP_FADVISE (from 5.6)
- Path based syscalls with async impersonation (from 5.6)
  - IORING_OP_OPENAT2, IORING_OP_STATX
  - Using IORING_REGISTER_PERSONALITY for impersonation
  - IORING_OP_UNLINKAT, IORING_OP_RENAMEAT (from 5.10)
  - IORING_OP_MKDIRAT, IORING_OP_SYMLINKAT, IORING_OP_LINKAT (from 5.15)
  - IORING_OP_SETXATTR, IORING_OP_GETXATTR (from 5.19)

io-uring for Samba (Part 2)

- Between userspace and socket (and also filesystem) (from 5.8)
  - IORING_OP_SENDMSG, IORING_OP_RECVMSG
  - Improved MSG_WAITALL support (5.12, backported to 5.11, 5.10)
  - Maybe using IOSQE_ASYNC in order to avoid inline memcpy
  - IORING_OP_SPLICE, IORING_OP_TEE
  - IORING_OP_SENDMSG_ZC, zero copy with an extra completion (from 6.1)
  - IORING_OP_GET_BUF, under discussion to replace IORING_OP_SPLICE

vfs_io_uring in Samba 4.12 (2020)

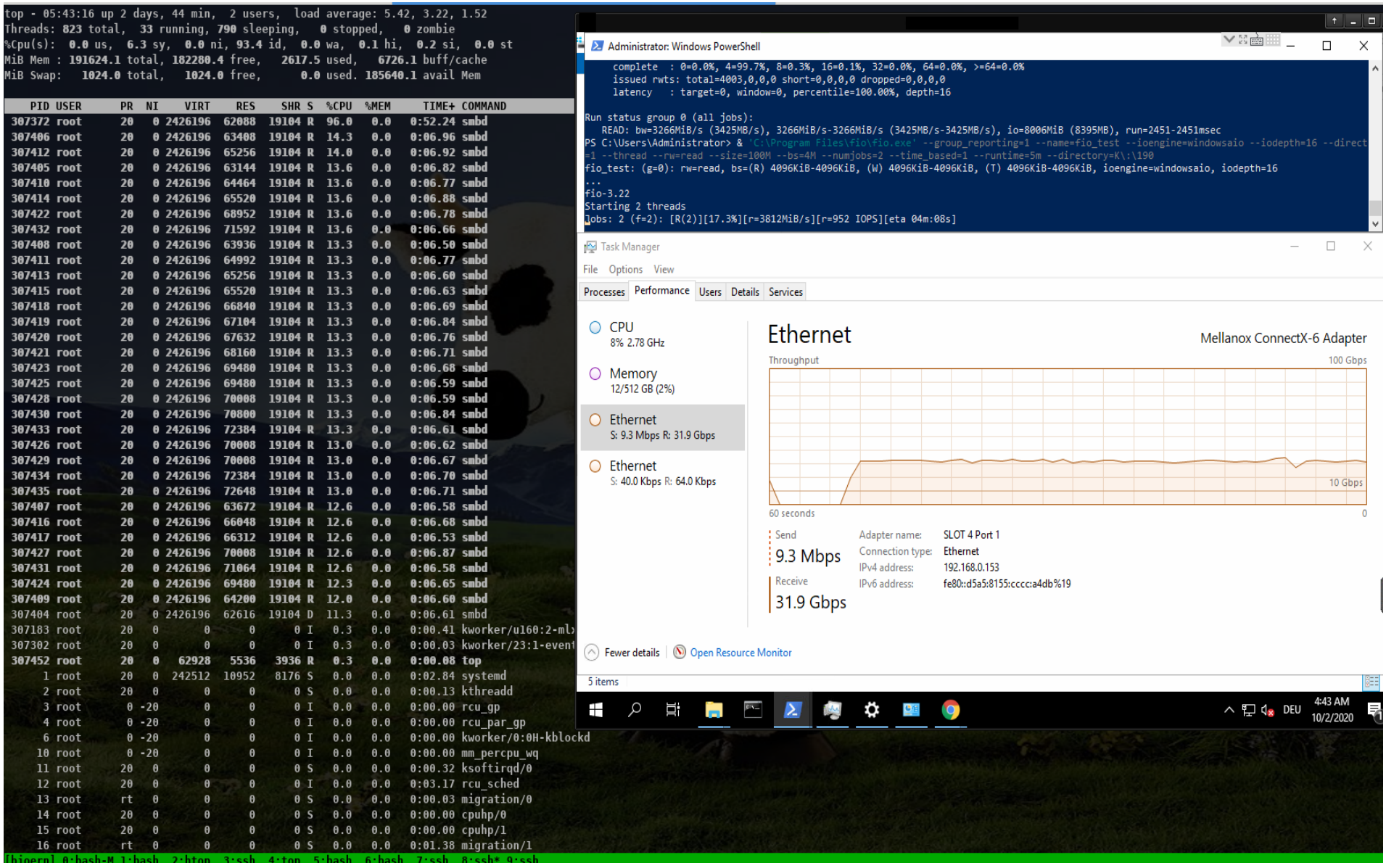
- With Samba 4.12 we added "io_uring" vfs module
  - For now it only implements SMB_VFS_PREAD,PWRITE,FSYNC_SEND/RECV
  - It has less overhead than our pthreadpool default implementations
  - I was able to speed up a smbclient 'get largefile /dev/null'
    - Using against smbd on loopback
    - The speed changes from 2.2GBytes/s to 2.7GBytes/s
- The improvement only happens by avoiding context switches
  - But the data copying still happens:
    - From/to a userspace buffer to/from the filesystem/page cache
  - The data path between userspace and socket is completely unchanged
  - For both cases the cpu is mostly busy with memcpy

Performance research (SMB2 Read)

- In October 2020 I was able to do some performance research
  - With 100GBit/s interfaces and two NUMA nodes per server.
- At that time I focussed on the SMB2 Read performance only
  - We had limited time on the given hardware
  - We mainly tested with fio.exe on a Windows client
  - Linux kernel 5.8.12 on the server
- More verbose details can be found here:
  - https://lists.samba.org/archive/samba-technical/2020-October/135856.html

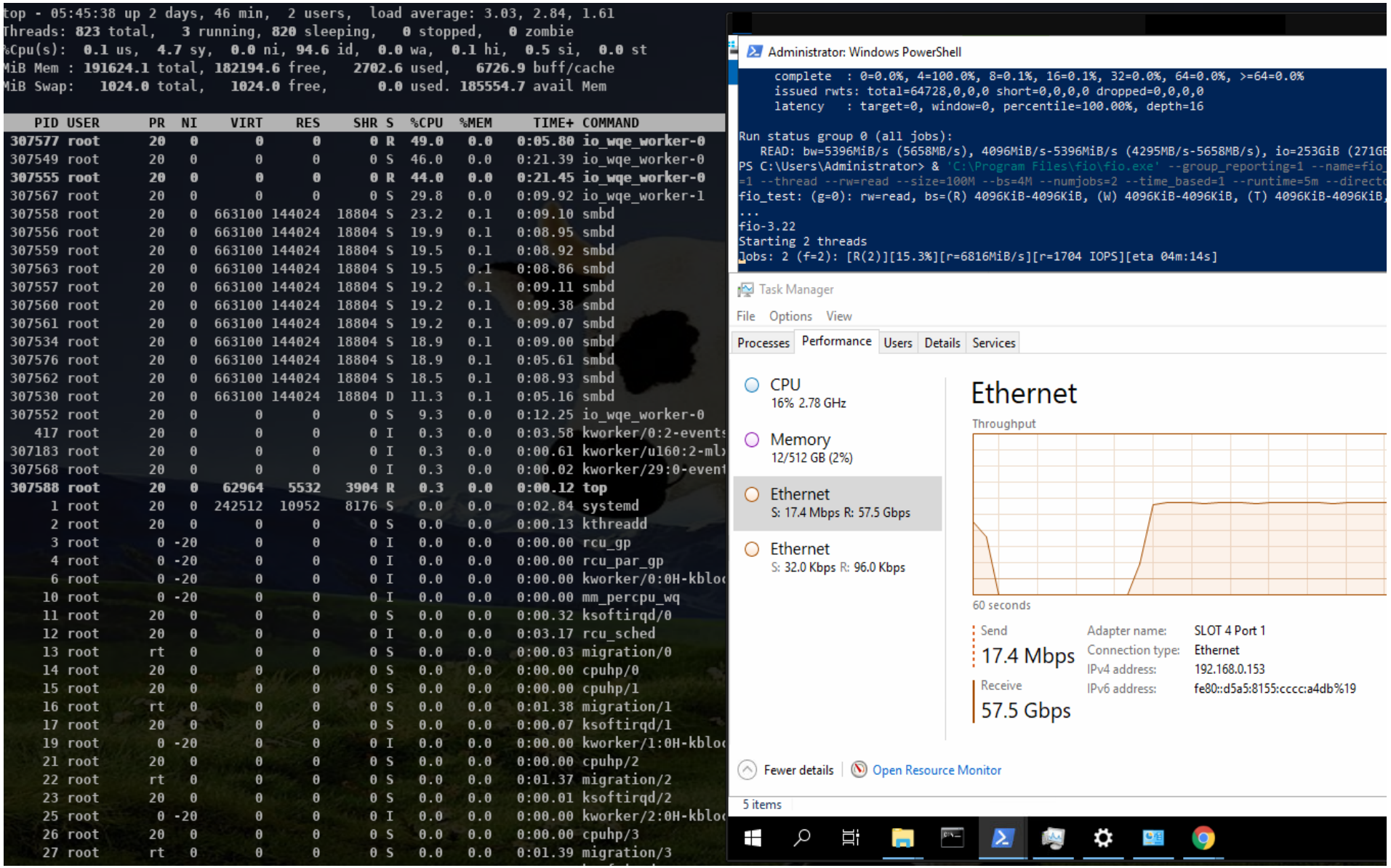
Performance with MultiChannel, sendmsg()

4 connections, ~3.8 GBytes/s, bound by >500% cpu in total, sendmsg() takes up to 0.5 msecs



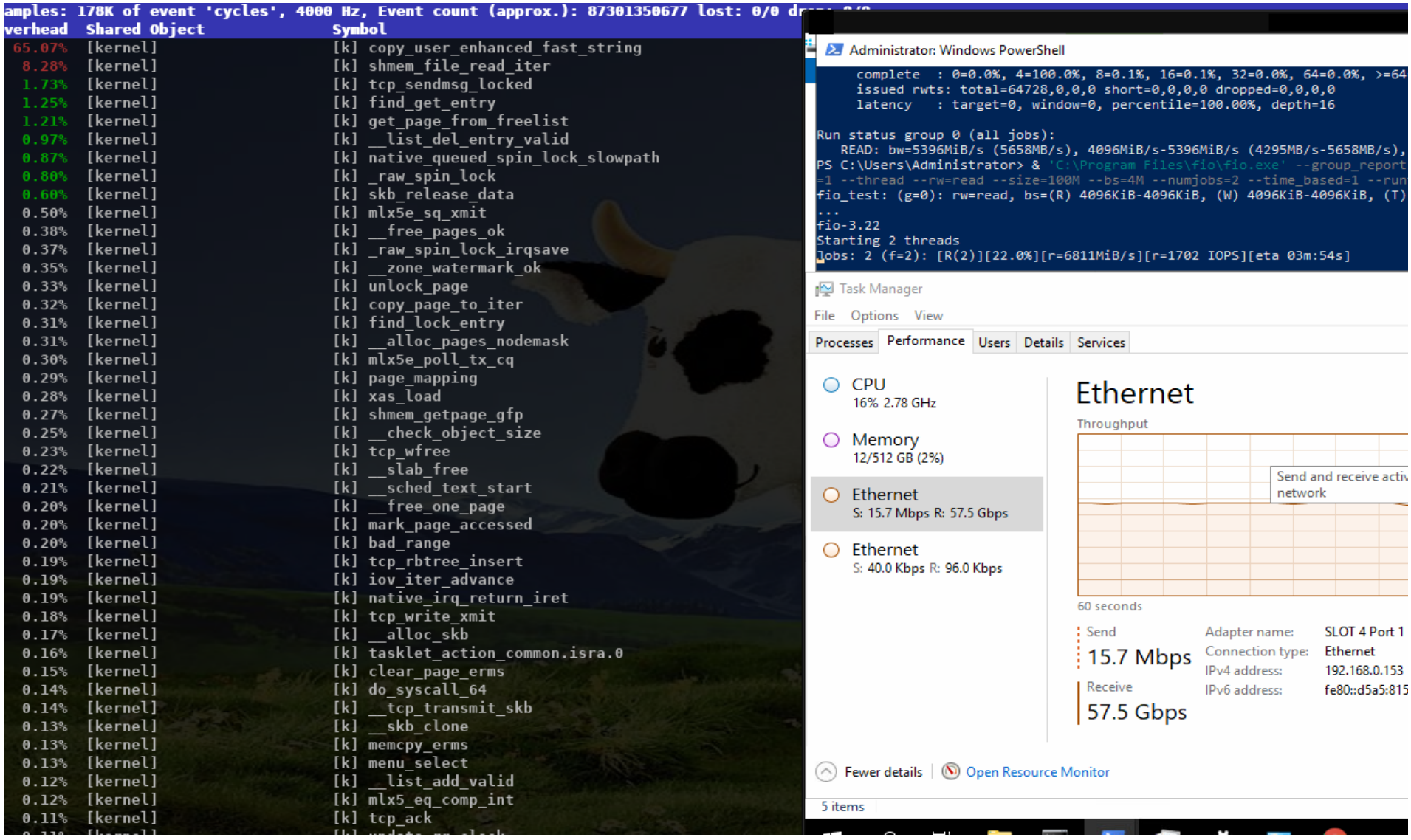
IORING_OP_SENDMSG (Part1)

4 connections, ~6.8 GBytes/s, smbd only uses ~11% cpu, (io_wqe_work ~50% cpu) per connection, we still use >300% cpu in total



IORING_OP_SENDMSG (Part2)

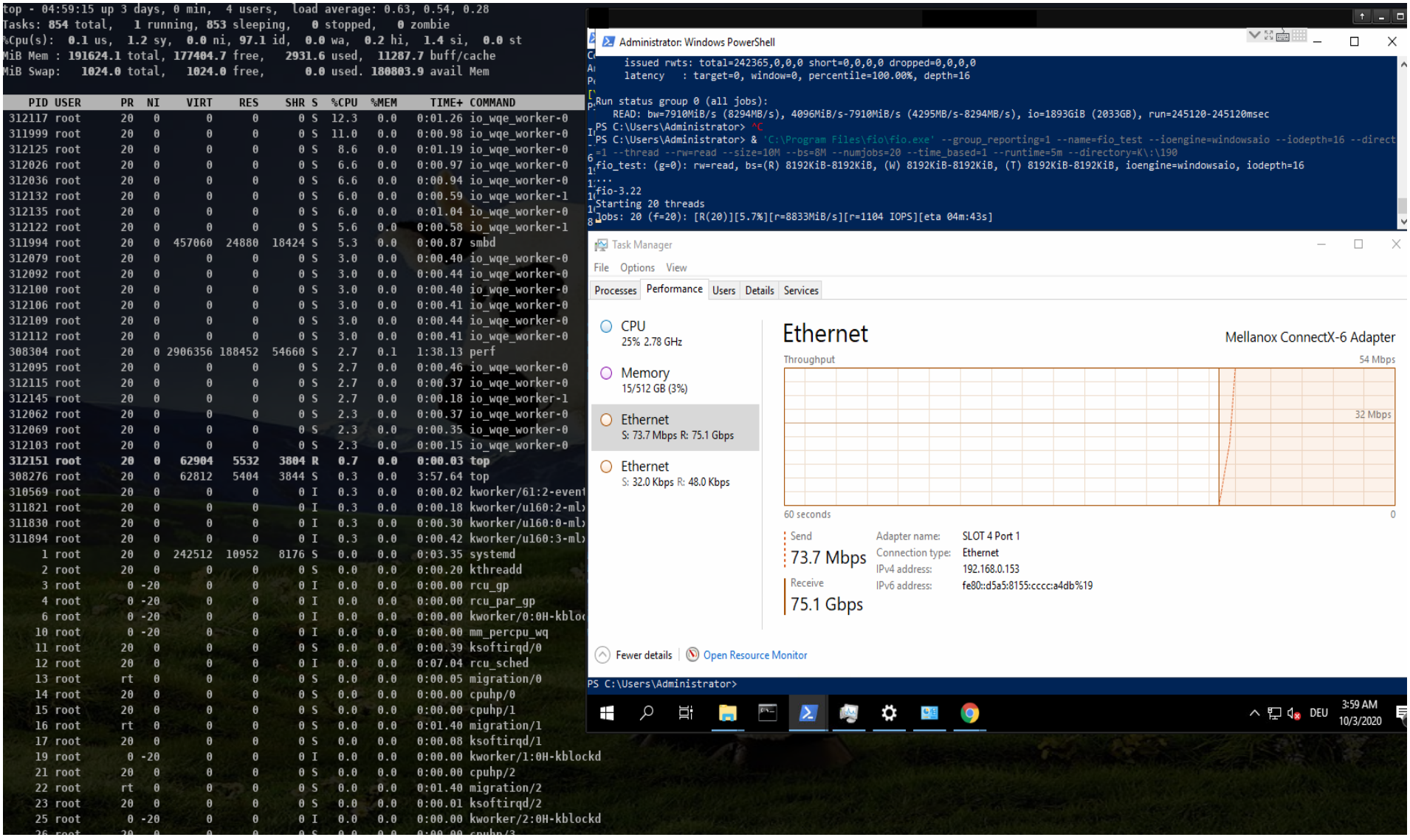
The major problem still exists, memory copy done by copy_user_enhanced_fast_string()



IORING_OP_SENDMSG + IORING_OP_SPLICE (Part1)

16 connections, ~8.9 GBytes/s, smbd ~5% cpu, (io_wqe_work 3%-12% cpu filesystem->pipe->socket), only ~100% cpu in total.

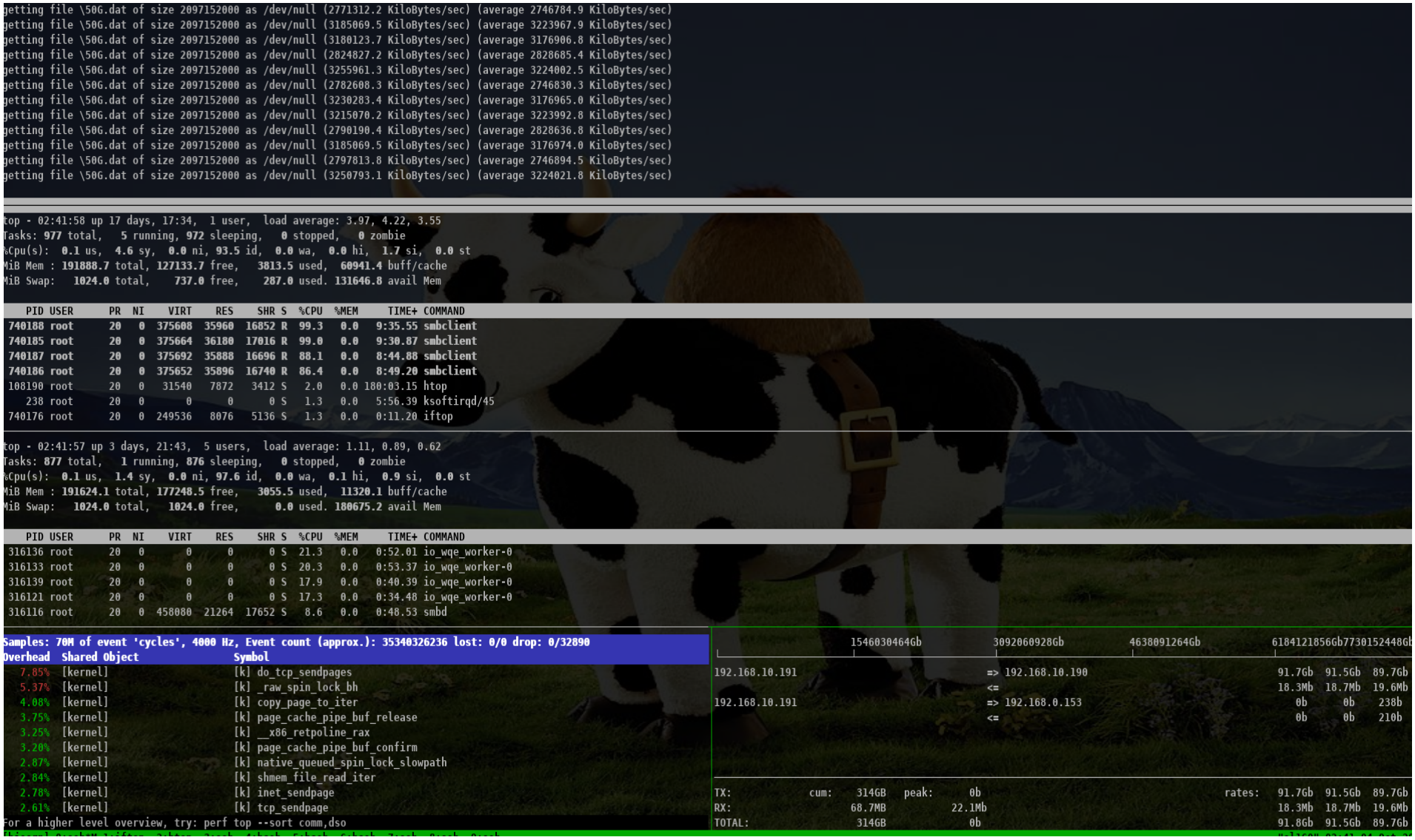
The Windows client was still the bottleneck with "Set-SmbClientConfiguration -ConnectionCountPerRssNetworkInterface 16"



smbclient IORING_OP_SENDMSG/SPLICE (network)

4 connections, ~11 GBytes/s, smbd 8.6% cpu, with 4 io_wqe_work threads (pipe to socket) at ~20% cpu each.

smbclient is the bottleneck here too



smbclient IORING_OP_SENDMSG/SPLICE (loopback)

8 connections, ~22 GBytes/s, smbd 22% cpu, with 4 io_wqe_work threads (pipe to socket) at ~22% cpu each.

smbclient is the bottleneck here too, it triggers the memory copy done by copy_user_enhanced_fast_string()

```
getting file \50G.dat of size 2097152000 as /dev/null (3075074.6 KiloBytes/sec) (average 2888801.8 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2942528.3 KiloBytes/sec) (average 2943679.6 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2719787.2 KiloBytes/sec) (average 2841637.3 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2951008.2 KiloBytes/sec) (average 2879437.6 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2801641.2 KiloBytes/sec) (average 2739170.8 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3107738.5 KiloBytes/sec) (average 2958064.5 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2694736.5 KiloBytes/sec) (average 2714142.3 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2860334.8 KiloBytes/sec) (average 2733460.0 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3117198.9 KiloBytes/sec) (average 2890262.3 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3047618.6 KiloBytes/sec) (average 2944358.1 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3098335.4 KiloBytes/sec) (average 2741473.6 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2741632.8 KiloBytes/sec) (average 2840912.6 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3002932.1 KiloBytes/sec) (average 2880254.5 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3126717.1 KiloBytes/sec) (average 2959135.8 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3088989.0 KiloBytes/sec) (average 2891536.4 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2515970.2 KiloBytes/sec) (average 2731748.8 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2171791.9 KiloBytes/sec) (average 2709204.0 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2921540.2 KiloBytes/sec) (average 2944203.8 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3093655.1 KiloBytes/sec) (average 2743720.7 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3093655.1 KiloBytes/sec) (average 2842525.3 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3007341.7 KiloBytes/sec) (average 2881088.4 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3107738.5 KiloBytes/sec) (average 2960079.4 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3136293.6 KiloBytes/sec) (average 2893072.3 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2752687.8 KiloBytes/sec) (average 2731898.3 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3084336.9 KiloBytes/sec) (average 2945095.8 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2745308.0 KiloBytes/sec) (average 2709462.2 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3117198.9 KiloBytes/sec) (average 2746070.8 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3117198.9 KiloBytes/sec) (average 2844253.7 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2563203.7 KiloBytes/sec) (average 2878659.8 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2519064.9 KiloBytes/sec) (average 2956651.4 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3093655.1 KiloBytes/sec) (average 2894340.3 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2828728.9 KiloBytes/sec) (average 2732566.5 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2771312.2 KiloBytes/sec) (average 2709897.3 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3131498.0 KiloBytes/sec) (average 2846041.8 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3131498.0 KiloBytes/sec) (average 2748470.0 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2595690.4 KiloBytes/sec) (average 2942472.7 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3038575.2 KiloBytes/sec) (average 2957176.0 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2976743.8 KiloBytes/sec) (average 2879300.8 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (3038575.2 KiloBytes/sec) (average 2895262.7 KiloBytes/sec)
getting file \50G.dat of size 2097152000 as /dev/null (2824827.2 KiloBytes/sec) (average 2733199.6 KiloBytes/sec)
```

```
top - 04:00:58 up 4 days, 23:02,  6 users,  load average: 9.15, 3.56, 1.44
Tasks: 917 total,  14 running, 903 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us, 11.2 sy,  0.0 ni, 86.1 id,  0.0 wa,  0.2 hi,  2.1 si,  0.0 st
MiB Mem : 191624.1 total, 176925.4 free,   3316.7 used,  11382.0 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used. 180403.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 322763 root      20   0  376228  36620  17364 R  82.5   0.0   1:26.20 smbclient
 322764 root      20   0  368036  28192  17120 R  81.5   0.0   1:26.18 smbclient
 322765 root      20   0  368040  28516  17164 R  80.1   0.0   1:25.16 smbclient
 322760 root      20   0  376244  36740  17468 R  79.8   0.0   1:23.73 smbclient
 322762 root      20   0  376236  36480  17220 R  79.8   0.0   1:24.42 smbclient
 322761 root      20   0  376248  28920  17292 R  79.5   0.0   1:24.74 smbclient
 322766 root      20   0  368040  28540  17464 R  79.5   0.0   1:25.93 smbclient
 322759 root      20   0  376140  36484  17312 R  78.1   0.0   1:24.31 smbclient
 322782 root      20   0       0      0      0 R  23.8   0.0   0:14.04 io_wqe_worker-0
 322827 root      20   0       0      0      0 S  23.5   0.0   0:12.77 io_wqe_worker-0
 322802 root      20   0       0      0      0 S  22.8   0.0   0:14.36 io_wqe_worker-0
 322830 root      20   0       0      0      0 S  22.8   0.0   0:12.96 io_wqe_worker-0
 322772 root      20   0  458260  21408  17596 R  22.5   0.0   0:22.45 smbd
 322796 root      20   0       0      0      0 S  22.2   0.0   0:14.08 io_wqe_worker-0
 322808 root      20   0       0      0      0 S  21.5   0.0   0:14.13 io_wqe_worker-0
 322822 root      20   0       0      0      0 R  21.5   0.0   0:12.86 io_wqe_worker-0
 322818 root      20   0       0      0      0 S  19.2   0.0   0:12.71 io_wqe_worker-0
 318818 root      20   0  248476   6976   4980 S   9.3   0.0   1:31.29 iftop
 322833 root      20   0       0      0      0 R   5.3   0.0   0:02.78 io_wqe_worker-0
 322854 root      20   0       0      0      0 S   5.0   0.0   0:02.50 io_wqe_worker-0
 322842 root      20   0       0      0      0 S   4.6   0.0   0:02.70 io_wqe_worker-0
 322851 root      20   0       0      0      0 S   4.6   0.0   0:02.49 io_wqe_worker-0
 322860 root      20   0       0      0      0 S   4.6   0.0   0:02.54 io_wqe_worker-0
 322862 root      20   0       0      0      0 S   4.6   0.0   0:02.70 io_wqe_worker-0
 318730 root      20   0 3037104 172756  54344 S   4.3   0.1   1:49.89 perf
 322836 root      20   0       0      0      0 S   4.3   0.0   0:02.61 io_wqe_worker-0
 322839 root      20   0       0      0      0 S   4.3   0.0   0:02.77 io_wqe_worker-0
 322848 root      20   0       0      0      0 R   4.0   0.0   0:02.52 io_wqe_worker-0
 322865 root      20   0       0      0      0 S   4.0   0.0   0:02.68 io_wqe_worker-0
 322868 root      20   0       0      0      0 S   4.0   0.0   0:02.66 io_wqe_worker-0
 322887 root      20   0       0      0      0 S   4.0   0.0   0:02.57 io_wqe_worker-0
 322845 root      20   0       0      0      0 S   3.6   0.0   0:02.50 io_wqe_worker-0
 322856 root      20   0       0      0      0 S   3.6   0.0   0:02.33 io_wqe_worker-0
 322858 root      20   0       0      0      0 S   3.6   0.0   0:02.52 io_wqe_worker-0
```

```
Samples: 30M of event 'cycles', 1000 Hz, Event count (approx.): 526705509529 lost: 0/0 drop: 0/0
Overhead  Shared Object                Symbol
  51.14%  [kernel]                     [k] copy_user_enhanced_fast_string
   6.48%  [kernel]                     [k] native_queued_spin_lock_slowpath
   3.38%  [kernel]                     [k] tpacket_rcv
   1.78%  [kernel]                     [k] do_tcp_sendpages
   1.28%  [kernel]                     [k] _raw_spin_lock_bh
   1.21%  [kernel]                     [k] prb_fill_curr_block.isra.0
   1.01%  [kernel]                     [k] _raw_spin_lock
   0.92%  [kernel]                     [k] copy_page_to_iter
   0.89%  [kernel]                     [k] skb_release_data
   0.89%  [kernel]                     [k] __check_object_size
For a higher level overview, try: perf top --sort comm,dso
```

```
                    1575537920Gb          3151075840Gb          4726614016Gb          6302151680Gb7877689344Gb
127.0.0.1                            => 127.0.0.1                                      181Gb    181Gb    180Gb
                                     <=                                                  0b       0b       0b

TX:             cum:   22642GB  peak:   6.59Gb                          rates:    181Gb    181Gb    180Gb
RX:                       0B            0b                                          0b       0b       0b
TOTAL:                 22642GB          6.59Gb                                     181Gb    181Gb    180Gb
```

More loopback testing on brand new hardware

- Recently I re-did the loopback read tests IORING_OP_SENDMSG/SPLICE (from /dev/shm/)
  - 1 connection, ~10-13 GBytes/s, smbd 7% cpu, with 4 iou-wrk threads at 7%-50% cpu.
  - 4 connections, 24-30 GBytes/s, smbd 18% cpu, with 16 iou-wrk threads at 3%-35% cpu.
- I also implemented SMB2 writes with IORING_OP_RECVMSG/SPLICE (tested to /dev/null)
  - 1 connection, ~7-8 GBytes/s, smbd 5% cpu, with 3 io-wrk threads at 1%-20% cpu.
  - 4 connections, ~10 GBytes/s, smbd 15% cpu, with 12 io-wrk threads at 1%-20% cpu.
- I tested with a Linux Kernel 5.13
  - In both cases the bottleneck is clearly on the smbclient side
  - We could apply similar changes to smbclient and add true multichannel support
  - It seems that the filesystem->pipe->socket path is much better optimized

The road to upstream (TEVENT_FD_ERROR)

- We need support for TEVENT_FD_ERROR in order to monitor errors
  - When using IORING_OP_SEND,RECVMSG we still want to notice errors
  - This is the main merge request:
  - https://gitlab.com/samba-team/samba/-/merge_requests/2793
  - This merge request converts Samba to use TEVENT_FD_ERROR:
  - https://gitlab.com/samba-team/samba/-/merge_requests/2885
  - (It also simplifies other places in the code without io_uring)

The road to upstream (samba_io_uring abstraction 1)

API glue to tevent:

```
void samba_io_uring_ev_register(void);

const struct samba_io_uring_features *samba_io_uring_system_features(void);

struct samba_io_uring *samba_io_uring_ev_context_get_ring(struct tevent_context *ev);

const struct samba_io_uring_features *samba_io_uring_get_features(
                                        const struct samba_io_uring *ring);

ev = tevent_context_init_byname(mem_ctx, "samba_io_uring_ev");
```

- samba_io_uring abstraction factored out of vfs_io_uring:
  - samba_io_uring_ev_hybrid tevent backend (glued on epoll backend)
  - It means every layer getting the tevent_context can use io_uring
  - No #ifdef's just checking if the required features are available

The road to upstream (samba_io_uring abstraction 2)

generic submission/completion api:

```
void samba_io_uring_completion_prepare(struct samba_io_uring_completion *completion,
            void (*completion_fn)(struct samba_io_uring_completion *completion,
                                  void *completion_private,
                                  const struct io_uring_cqe *cqe),
            void *completion_private);

void samba_io_uring_submission_prepare(struct samba_io_uring_submission *submission,
            void (*submission_fn)(struct samba_io_uring *ring,
                                  struct samba_io_uring_submission *submission,
                                  void *submission_private),
            void *submission_private,
            struct samba_io_uring_completion *completion);

struct io_uring_sqe *samba_io_uring_submission_sqe(struct samba_io_uring_submission *
     submission);

size_t samba_io_uring_queue_submissions(struct samba_io_uring *ring,
                                        struct samba_io_uring_submission *submission);
```

- Using it ...
  - convert vfs_io_uring
  - use it in smb2_server.c
  - In future use it in other performance critical places too.

The road to upstream (smb2_server.c)

- Refactoring of smb2_server.c
  - add optional IORING_OP_SENDMSG, IORING_OP_RECVMSG support
- There are structural problems with splice from a file
  - I had a discussion with the Linux developers about it:
  - The page content from the page cache may change unexpectetly
  - https://lists.samba.org/archive/samba-technical/2023-February/thread.html#137945
  - We may not able to use IORING_OP_SENDMSG/SPLICE by default
  - Maybe IORING_OP_RECVMSG/SPLICE is possible
- At least we can have only 1 one copy instead of two:
  - IORING_OP_SENDMSG_ZC is able to avoid copying to the socket
    - we get an extra completion once the buffers are not needed anymore
  - This gives good results, between with and without IORING_OP_SENDMSG/SPLICE
  - But I don't have numbers as it doesn't work on loopback
  - Within VM's improvement can be seen

Future Improvements

- I have a prototype for a native io_uring tevent backend:
  - The idea is to avoid epoll and only block in io_uring_enter()
  - But the semantics of IORING_OP_POLL_ADD,REMOVE are not useable
  - https://lists.samba.org/archive/samba-technical/2022-October/thread.html#137734
  - We may get an IORING_POLL_CANCEL_ON_CLOSE in future
  - And a usable IORING_POLL_LEVEL
- We can use io_uring deep inside of the smbclient code
  - The low layers can just use samba_io_uring_ev_context_get_ring()
  - And use if available without changing the whole stack

Questions? Feedback!

- Stefan Metzmacher, metze@samba.org
- https://www.sernet.com
- https://samba.plus

→ SerNet/SAMBA+ sponsor booth

Slides: https://samba.org/~metze/presentations/2023/SDC/