# The Evolution of File Descriptor Monitoring in Linux

## From select(2) to io_uring

Stefan Hajnoczi <stefanha@gmail.com>
FOSDEM 2021

# What is File Descriptor Monitoring?

API for determining when a file descriptor becomes ready to perform I/O

- *Is a client connecting to a listening socket?*
- *Has a new message arrived on a socket?*
- *Is it possible to write more data to a pipe?*

# Linux APIs

- select(2)
- poll(2)
- epoll(7)
- io_uring
- Also: fasync/SIGIO, Linux AIO

# Kernel Interface

These APIs are implemented using one* interface:

```
struct file_operations {
    __poll_t (*poll) (struct file *,
                      struct poll_table_struct *);
};
```

Set of events that are ready

* except fasync/SIGIO

# Linux File Descriptor Events

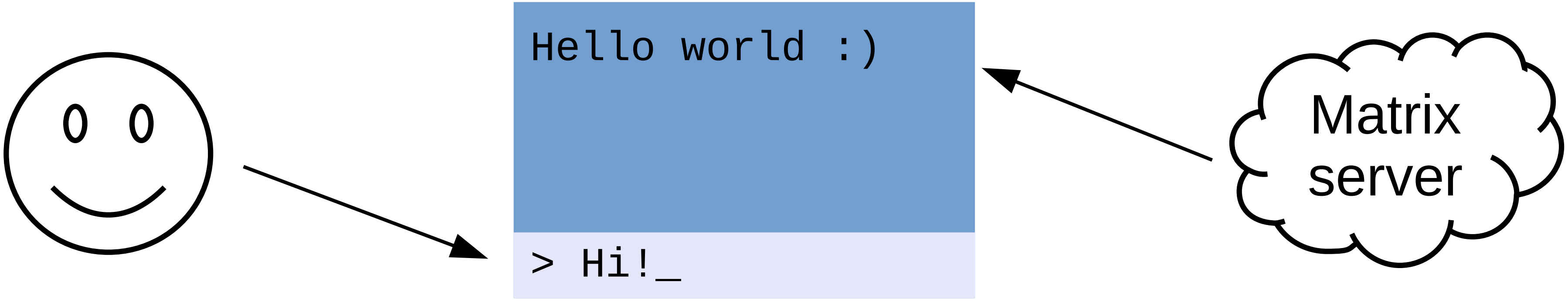
| | |
|---|---|
| EPOLLIN | Ready for read(2) |
| EPOLLOUT | Ready for write(2) |
| EPOLLRDHUP | Socket peer will not write anymore |
| EPOLLPRI | File-specific exceptional condition |
| EPOLLERR | Error or reader closed pipe |
| EPOLLHUP | Socket peer closed connection |

Plus rarely-used out-of-band events.

Spurious EPOLLIN is possible, use O_NONBLOCK

# Why use File Descriptor Monitoring?

Example: text-based Matrix chat client




We want to respond to user input from terminal *and* Matrix activity from network.

# Event-driven Architecture

Spawning a thread for each I/O task requires coordination and resources. Is there another way for I/O bound applications?

Needs file descriptor monitoring

```
while (running) {
    Event *event = next_event();
    handle(event);
}
```

# Where is it used?

- GUI applications (Qt, GTK, etc)
- Servers (nginx, etc)
- “Thread-per-core architecture”
- Sometimes just to add timeouts or cancellation to blocking syscalls

# select(2)

Total bits set

Highest fd number plus 1

```
int pselect(int nfds,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            const struct timespec *timeout,
            const sigset_t *sigmask);
```

Input:
Set bit *n*
to monitor
fd number *n*

Output:
Bit *n* is set
if fd number *n*
is ready

# select(2) Quirks

- Inefficient for sparse bitmaps, lots of scanning

  00000000000000000000000001 ← fd 25

- FD_SETSIZE limit is 1024 on Linux (glibc)

  Can’t use select(2) if fd is larger.

# Associating Application Objects

select(2) does not make it easy to locate the corresponding application object for an fd in the general case.

Hard coded: ✓

```
if (FD_ISSET(tty_fd))
    read_tty_input();
```

General case: ✗

```
for (i=0; i<nfds; i++)
    if (FD_ISSET(i))
        obj->fd_ready();
```

# poll(2)

Ready event count

```
int ppoll(struct pollfd *fds, nfds_t nfds,
          const struct timespec *tmo_p,
          const sigset_t *sigmask);

struct pollfd {
    int fd;
    short events;
    short revents;
};
```

Input: Event mask to monitor

Output: Ready event mask

# poll(2) compared to select(2)

- Number of fds no longer limited to 1024 ✓
- Dense fd list ✓
- Input not overwritten, can be reused next call ✓
- Easy application object lookup ✓

```
for (i = 0; i < nfds && ret > 0; i++)
    if (fds[i].revents) {
        obj[i]->fd_ready(); ret--;
    }
```

# epoll_ctl(7)

Created with epoll_create1(2)

EPOLL_CTL_ADD,
EPOLL_CTL_MOD,
or EPOLL_CTL_DEL

```
int epoll_ctl(int epfd, int op, int fd,
              struct epoll_event *event);
struct epoll_event {
    uint32_t events;
    epoll_data_t data;
};
```

Event mask to monitor

Application-specific value

# epoll_pwait2(2)

Number of ready fds

New in Linux 5.11

```
int epoll_pwait2(int epfd,
                 struct epoll_event * events,
                 int maxevents,
                 const struct timespec *timeout,
                 const sigset_t *sigmask);
```

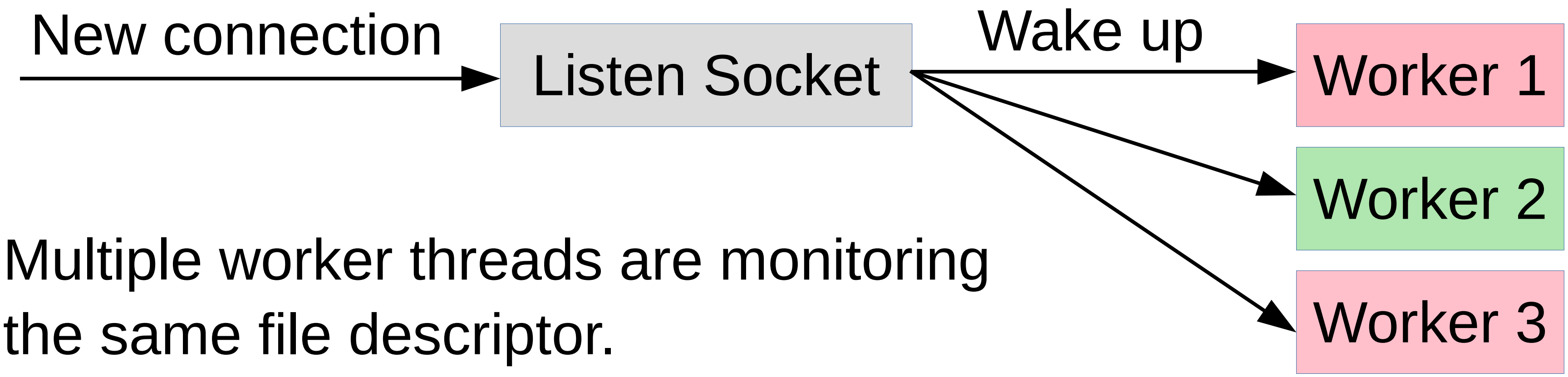
Array of events filled in by kernel

Size independent of number of monitored fds. Round-robin algorithm for fairness.

# epoll(7) flags

| | |
|---|---|
| EPOLLET | Edge-triggered mode |
| EPOLLONESHOT | Auto-disable on event |
| EPOLLEXCLUSIVE | Only wake one waiter,<br>solve Thundering Herd problem |

# Thundering Herd Problem

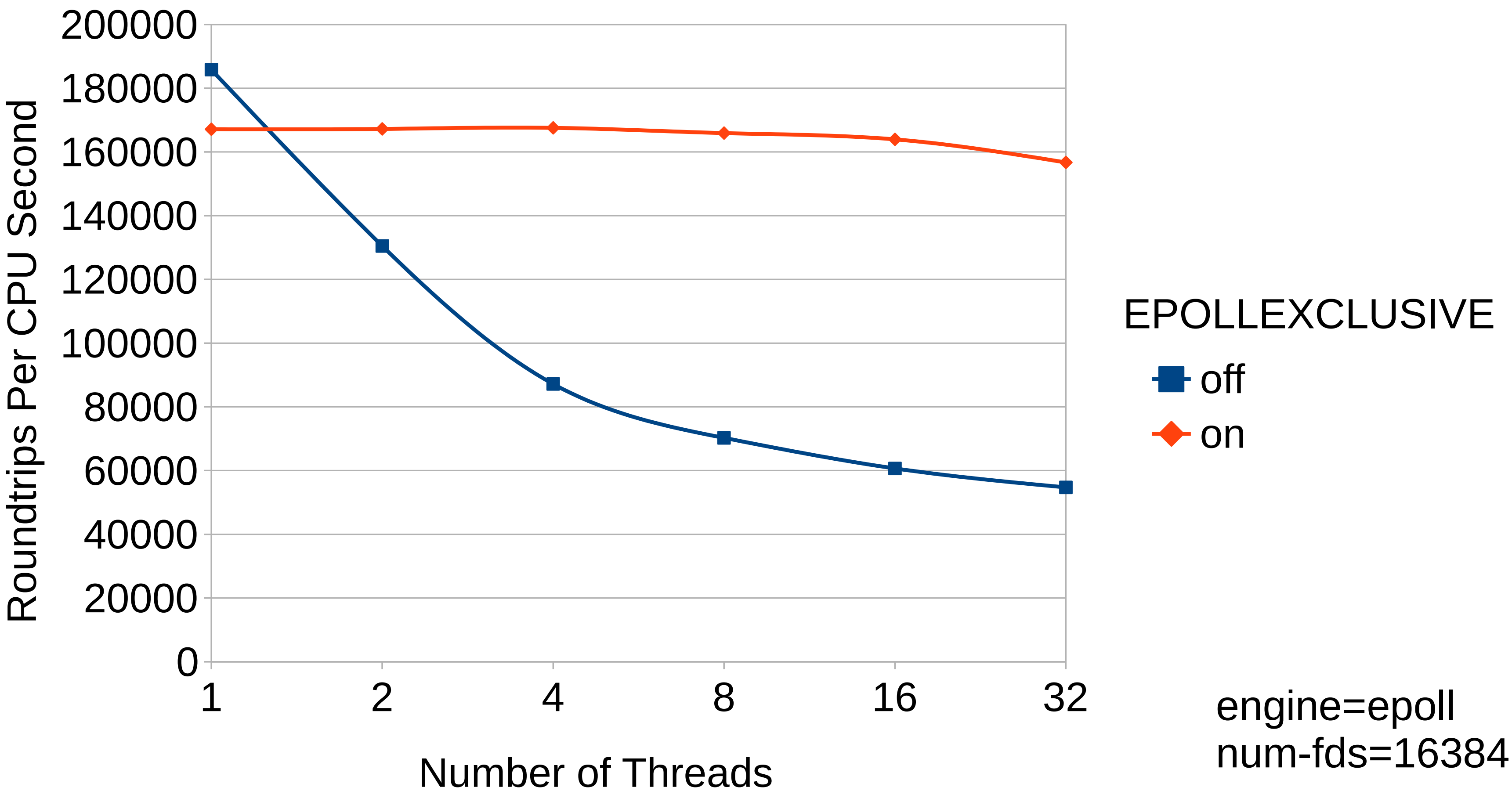



Multiple worker threads are monitoring the same file descriptor.

File descriptor monitoring wakes up all workers, but only one thread can handle the I/O.

CPU cycles are wasted waking up other workers.

# Thundering Herd CPU Efficiency




engine=epoll
num-fds=16384

# Stateless vs Stateful APIs

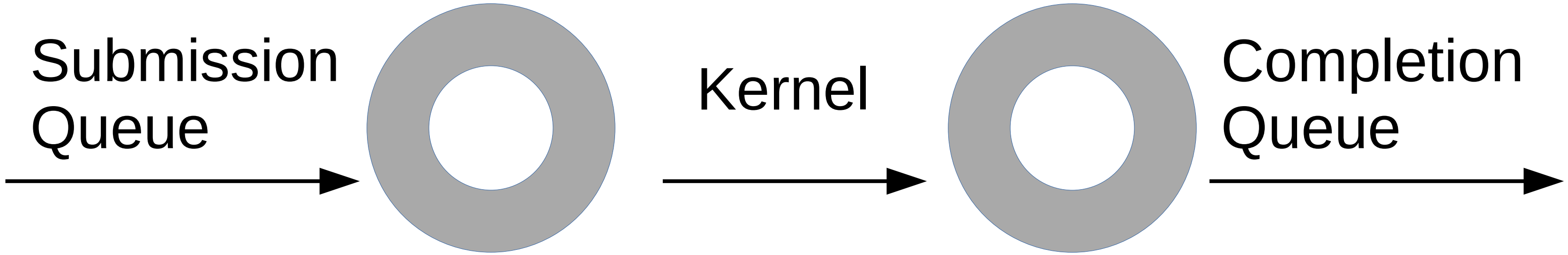
- select(2) and poll(2) are *stateless*
  - Kernel doesn’t remember which fds to monitor between system calls
- epoll(7) is *stateful*
  - epoll_pwait2(2) only collects results, **doesn’t need to set up fd monitoring each time**!

# epoll(7) is O(num_ready)

- select(2) required scanning O(max_fdnum)
- poll(2) required scanning O(nfds)
- epoll_pwait2(2) is O(num_ready)

→ App only sees fds that are ready!

# io_uring




```
int io_uring_enter(unsigned int fd,
                   unsigned int to_submit,
                   unsigned int min_complete,
                   unsigned int flags,
                   sigset_t *sig);
```

Number of reqs to submit → `to_submit`

Number of reqs to wait for → `min_complete`

# io_uring Operations

| | |
|---|---|
| IORING_OP_POLL_ADD | One-shot file descriptor monitoring |
| IORING_OP_POLL_REMOVE | Remove existing request |
| IORING_OP_EPOLL_CTL | Like epoll_ctl(2) |
| IORING_OP_TIMEOUT | Nanosecond timeout, can auto-cancel if other requests complete |

...and many more

# Liburing Example

```
struct io_uring_sqe *sqe;
sqe = io_uring_get_sqe(ring);
io_uring_prep_poll_add(sqe, fd, POLLIN);
io_uring_sqe_set_data(sqe, obj);
…
n = io_uring_submit_and_wait(ring, MAX_EVENTS);
io_uring_for_each_cqe(ring, head, cqe)
    handle(cqe->user_data);
io_uring_cq_advance(ring, n);
```

# io_uring Characteristics

- One system call to submit many fds ✓
- System call combines submission and completion ✓
- Userspace can busy wait on completions in mmapped completion queue, no system calls ✓
- Kernel can busy wait on submissions, no system calls ✓
- Much more: linked requests, registered fds and buffers, etc

# Net busy waiting

- Kernel busy waiting for sockets
- sysctl net.core.busy_poll=<microseconds>
- Busy wait in select(2), poll(2), epoll(7)
  - Avoids descheduling current task and idling CPU
- Busy waiting is useful when latency is important and there are dedicated CPUs available

# Is AIO the End of File Descriptor Monitoring?

Instead of splitting I/O tasks into fd monitoring and I/O steps, let the kernel perform I/O asynchronously with io_uring.

Monitoring approach:

```
wait_fd_readable(fd);

read(fd);

done_cb();
```

AIO approach:

```
async_read(fd, done_cb);
```

io_uring calls file_ops->poll() internally, fewer system calls needed

# Migrating to AIO

Applications and libraries are designed around file descriptor monitoring:

```
int monitor_fd(int fd, int events, ready_func ready_cb);
```

An AIO read interface looks like this:

```
int aio_read(int fd, off_t off, void *buf, size_t len,
             done_func done_cb);
```

Big change for existing code bases :(

See my blog for consequences on software ecosystem:
http://blog.vmsplice.net/2020/07/rethinking-event-loop-integration-for.html

# Other APIs

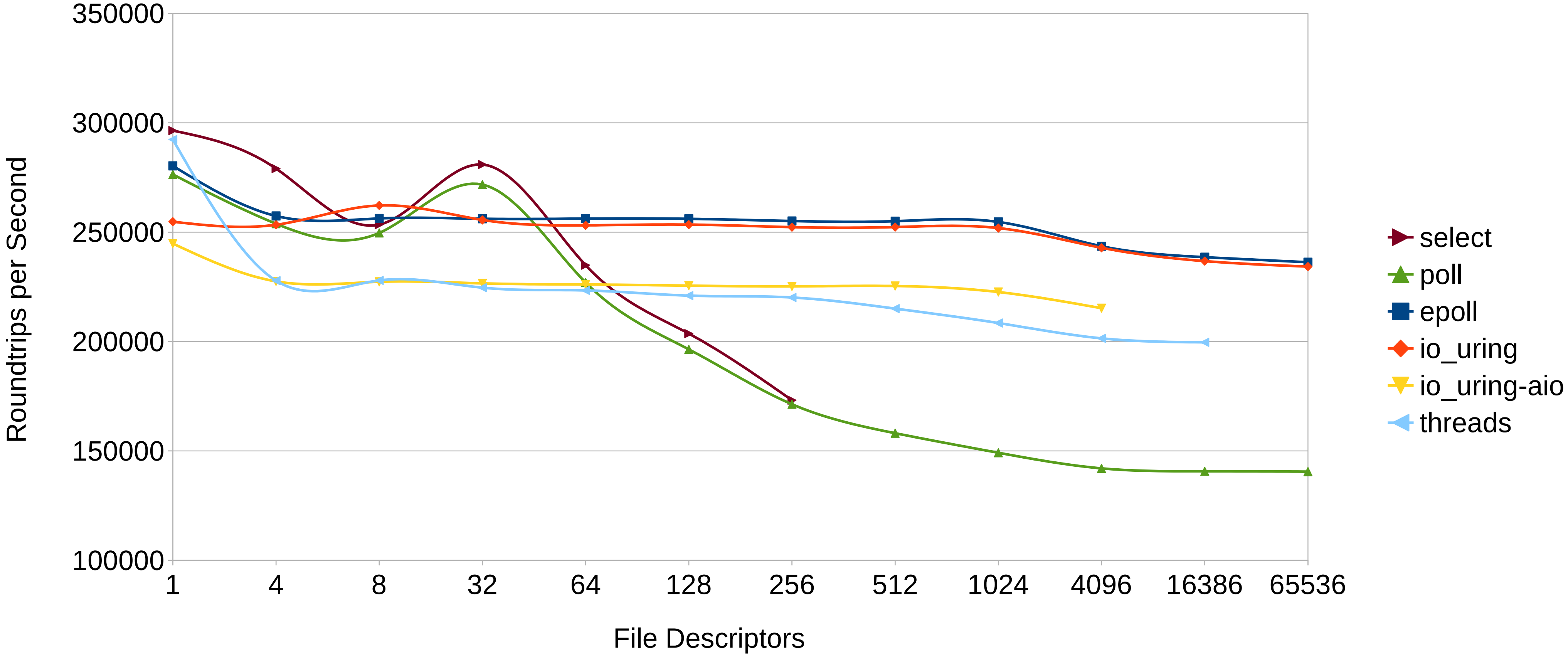
- fasync/SIGIO
  - Old signal-based mechanism
  - Rarely-used, programming with signals is tricky
- Linux AIO
  - Subset of io_uring functionality
  - Similar shared memory ring design

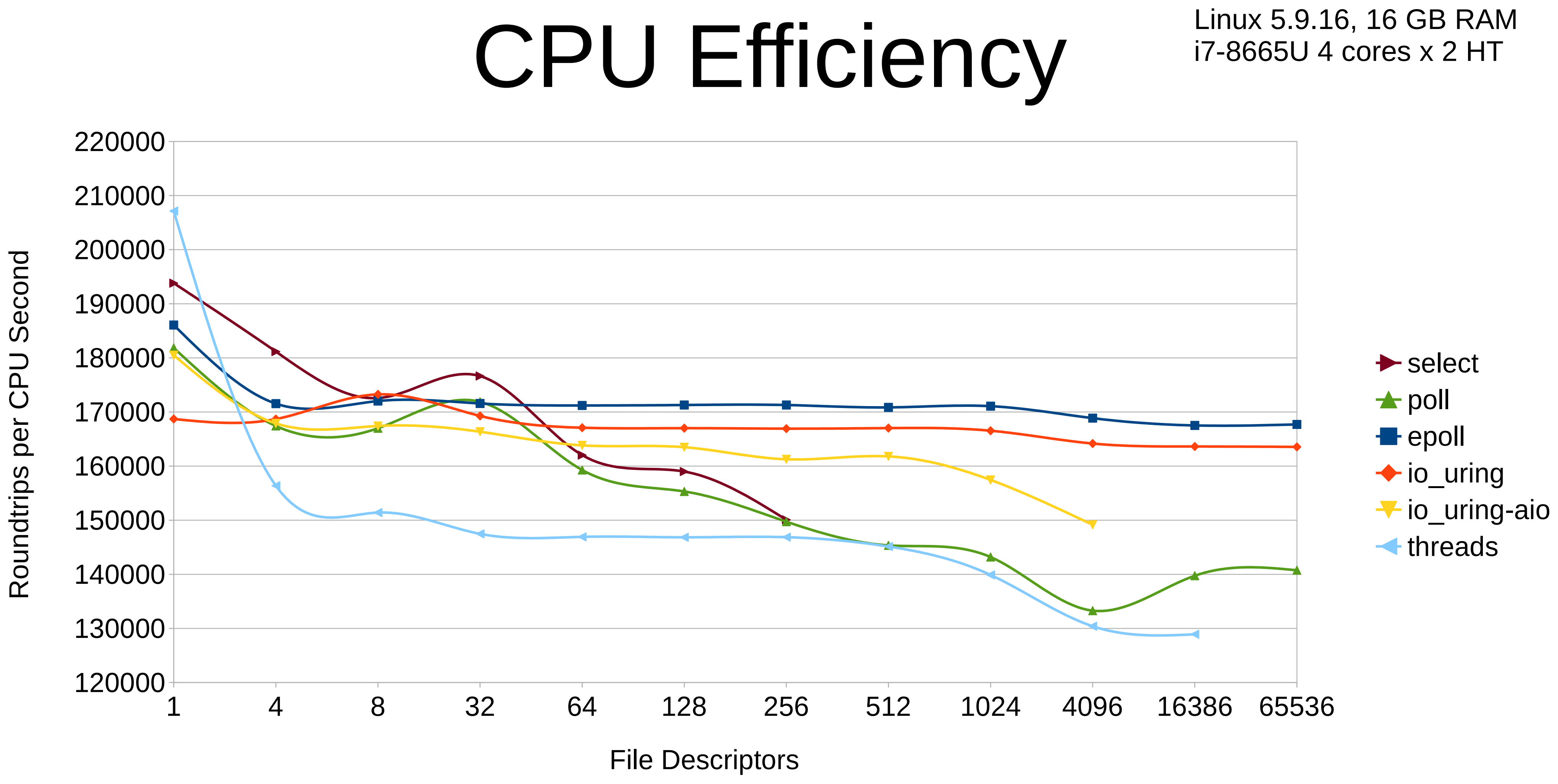
# fdmonbench

- Message is received on a random fd and is sent back
- No changes to set of monitored fds during benchmark
- Number of fds and number of receivers can be controlled
- https://github.com/stefanha/fdmonbench

# Scalability

Linux 5.9.16, 16 GB RAM
i7-8665U 4 cores x 2 HT




CPU Efficiency
Linux 5.9.16, 16 GB RAM
i7-8665U 4 cores x 2 HT
Roundtrips per CPU Second
220000
210000
200000
190000
180000
170000
160000
150000
140000
130000
120000
1
4
8
32
64
128
256
512
1024
4096
16386
65536
File Descriptors
select
poll
epoll
io_uring
io_uring-aio
threads

# API Summary

| API | POSIX? | Herd? | Complexity | Comments |
|---|---|---|---|---|
| select(2) | ✓ | ✗ | O(max_fdnum) | For small tasks |
| poll(2) | ✓ | ✗ | O(nfds) | For portability |
| epoll(7) | ✗ | ✓ | O(num_ready) | Popular today |
| io_uring | ✗ | ✓ | O(num_ready) | Popular tomorrow? |
| Linux AIO | ✗ | ✗ | O(num_ready) | io_uring fallback |

# Thank You

https://blog.vmsplice.net/

stefanha on IRC

@stefanha:matrix.org

stefanha@gmail.com