Epoll Kernel Performance Improvements

Open Source Summit – July 2019. Tokyo, Japan.

Davidlohr Bueso, SUSE Labs.

We adapt. You succeed.

Agenda (40 minutes).

1. Introduction.

2. Epoll Internal Architecture.

3. Upstreamed Performance Work.

4. Other Performance Work.

5. Benchmarking Epoll.

Introduction

"… monitoring multiple files to see if IO is possible on any of them..."

- man 7 epoll

Introduction

"… monitoring multiple files to see if IO is possible on any of them..."

- man 7 epoll

- epoll_create(2) – fd new epoll *instance*.
- epoll_ctl(2) – manage file descriptors regarding the *interested-list*.

Introduction

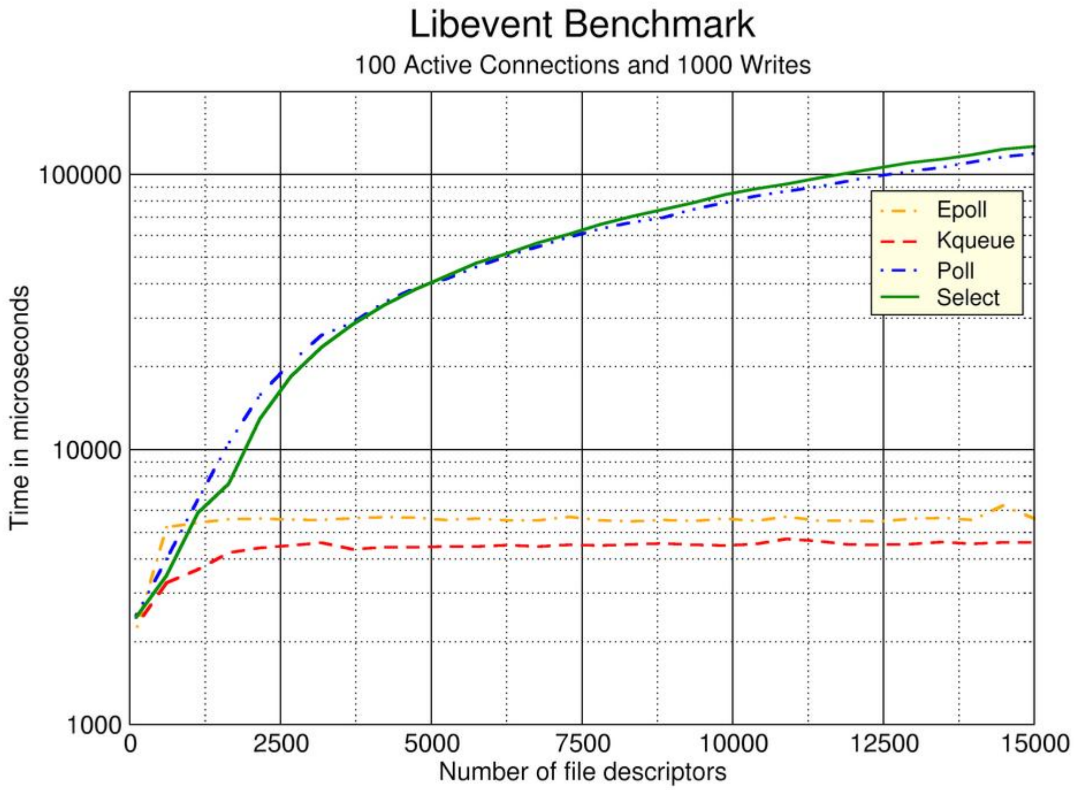
"… monitoring multiple files to see if IO is possible on any of them..."

- man 7 epoll

- epoll_create(2) – fd new epoll *instance*.
- epoll_ctl(2) – manage file descriptors regarding the *interested-list*.
- epoll_wait(2) – main workhorse, block tasks until IO becomes available.

Introduction

- Epoll scalability is better than it’s (Linux) rivals.



Introduction

- Epoll scalability is better than it’s (Linux) rivals.

- How is this accomplished?
  - Separate setup and waiting phases.
  - Keeping kernel internal data structures.
- This results in:
  - Upon ready IO, select/poll are O(n), epoll is O(n_ready).
  - Do not have to pass description of the fds.
  - Epoll can monitor an unlimited amount of fds.

Introduction

“epoll is fundamentally broken”

–some people online

- Was not initially designed for multi-threading in mind.
- Special programming is needed to use epoll in an efficient and race free manner.
  - EPOLLEXCLUSIVE – Wakeup a single task (level-triggered). Avoid thundering herd problem.
  - EPOLLONESHOT – Disable fd after receiving an event. Must rearm.

Introduction

“epoll is fundamentally broken”

(threads A and B are waiting on epoll, LT)

1. Kernel: receives 4095 bytes of data
2. Kernel: Thread A is awoken (ie EPOLLEXCLUSIVE).
3. Thread A: finishes epoll_wait(2)
4. Kernel: receives 4 bytes of data
5. Kernel: wakes up Thread B.
6. Thread A: performs read(4096) and reads full buffer of 4096 bytes
7. Thread B: performs read(4096) and reads remaining 3 bytes of data

SUSE

Introduction

“epoll is fundamentally broken”

(threads A and B are waiting on epoll, LT)

1. Kernel: receives 4095 bytes of data
2. Kernel: Thread A is awoken (ie EPOLLEXCLUSIVE).
3. Thread A: finishes epoll_wait(2)
4. Kernel: receives 4 bytes of data
5. Kernel: wakes up Thread B
6. Thread A: performs read(4096) and reads full buffer of 4096 bytes
7. Thread B: performs read(4096) and reads remaining 3 bytes of data

Data is split across threads and can be reordered without serialization.
The correct solution is to use EPOLLONESHOT and re-arm.

Plenty of examples:
https://idea.popcount.org/2017-02-20-epoll-is-fundamentally-broken-12/

Introduction

“epoll is fundamentally broken”

–some people online

- Associates the file descriptor with the underlying kernel object.
  - Tied to the lifetime of the object, not the fd.
- Broken `fork/close(2)` semantics.
  - It is possible to receive events after closing the fd.
  - Must EPOLL_CTL_DEL the fd before closing.

Epoll Internal Architecture

(main) Data Structures

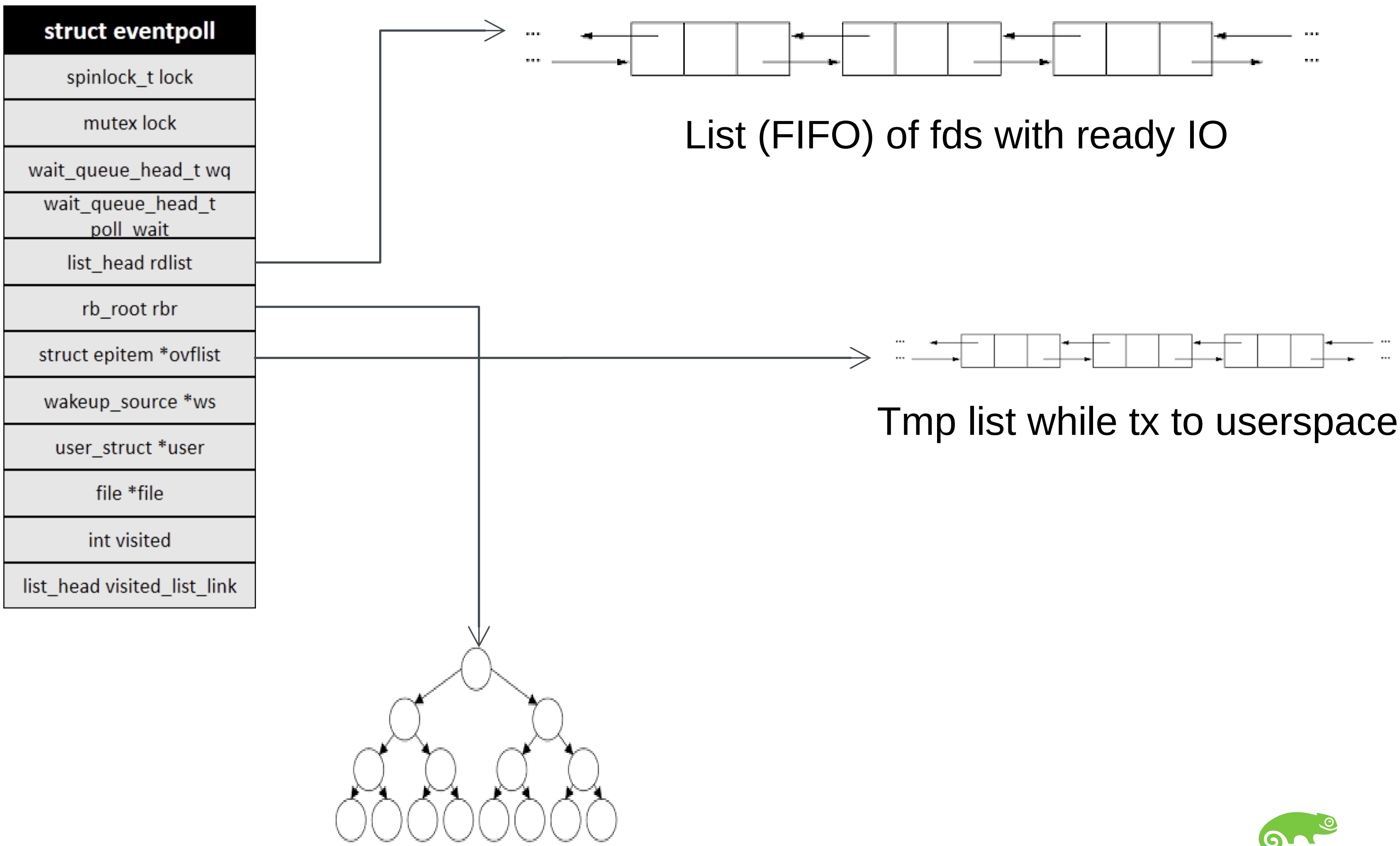
struct eventpoll
spinlock_t lock
mutex lock
wait_queue_head_t wq
wait_queue_head_t poll_wait
list_head rdlist
rb_root rbr
struct epitem *ovflist
wakeup_source *ws
user_struct *user
file *file
int visited
list_head visited_list_link

Instance from epoll_create()

struct epitem
rb_node rbn
list_head rdlink
epitem *next
epoll_filefd ffd
int nwait
list_head pwqlist
eventpoll *ep
list_head fllink
wakeup_source *ws
epoll_event event

Every fd in the interested-list

(main) Data Structures



Locking Rules

struct eventpoll
spinlock_t lock
mutex lock
wait_queue_head_t wq
wait_queue_head_t poll_wait
list_head rdlist
rb_root rbr
struct epitem *ovflist
wakeup_source *ws
user_struct *user
file *file
int visited
list_head visited_list_link

Mutex: serialization while transferring events to userspace

`copy_to_user` might block.

Protect `epoll_ctl(2)` operations, file exit, etc.

Spinlock: serialization inside IRQ context, cannot sleep.

Protects ready and *overflow* list manipulation.

(Must already hold the ep->mutex)

Locking Rules

struct eventpoll
spinlock_t lock
mutex lock
wait_queue_head_t wq
wait_queue_head_t poll_wait
list_head rdlist
rb_root rbr
struct epitem *ovflist
wakeup_source *ws
user_struct *user
file *file
int visited
list_head visited_list_link

Mutex: serialization while transferring events to userspace

copy_to_user might block.

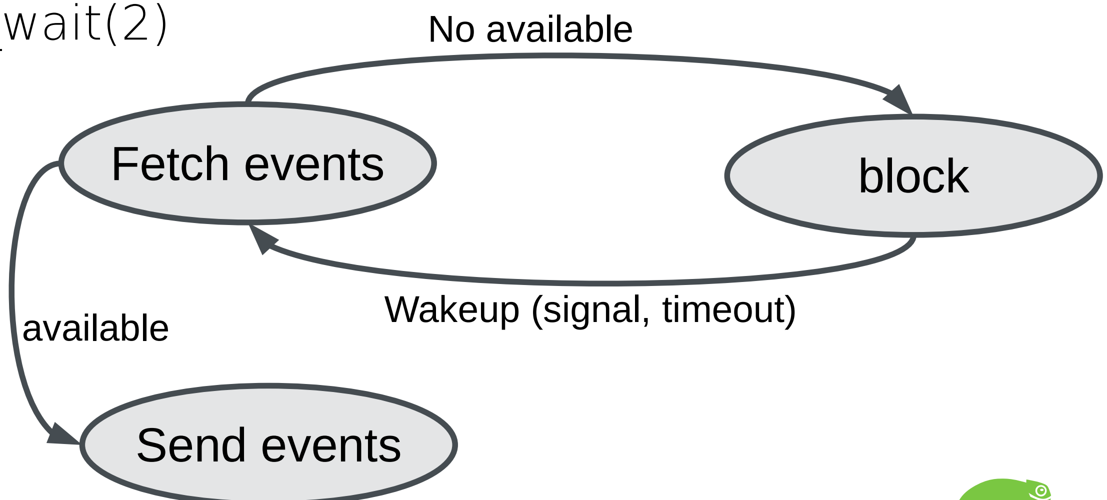
Protect epoll_ctl(2) operations, file exit, etc.

Spinlock: serialization inside IRQ context, cannot sleep.

Protects ready and *overflow* list manipulation.

(Must already hold the ep->mutex)

epoll_wait(2)



Locking Rules

- Both send events and wakeup callback need to operate on the ready list.
- When sending events, the overflow list kicks in.
  - Send events will run without the spinlock on a private list.

Locking Rules

```
spin_lock_irq(&ep->lock);
list_splice_init(&ep->rdllist, &txlist);
WRITE_ONCE(ep->ovflist, NULL);
spin_unlock_irq(&ep→lock);

<SEND_EVENTS>

spin_lock_irq(&ep→lock);
for (nepi = READ_ONCE(ep->ovflist); (epi = nepi) != NULL;
        nepi = epi->next, epi->next = EP_UNACTIVE_PTR)
            list_add(&epi->rdllink, &ep→rdllist);
WRITE_ONCE(ep->ovflist, EP_UNACTIVE_PTR);
list_splice(&txlist, &ep→rdllist);
spin_unlock_irq(&ep->lock);
```

ep_poll_callback():
Events that happen during this period are chained in ep->ovflist and requeued later on.

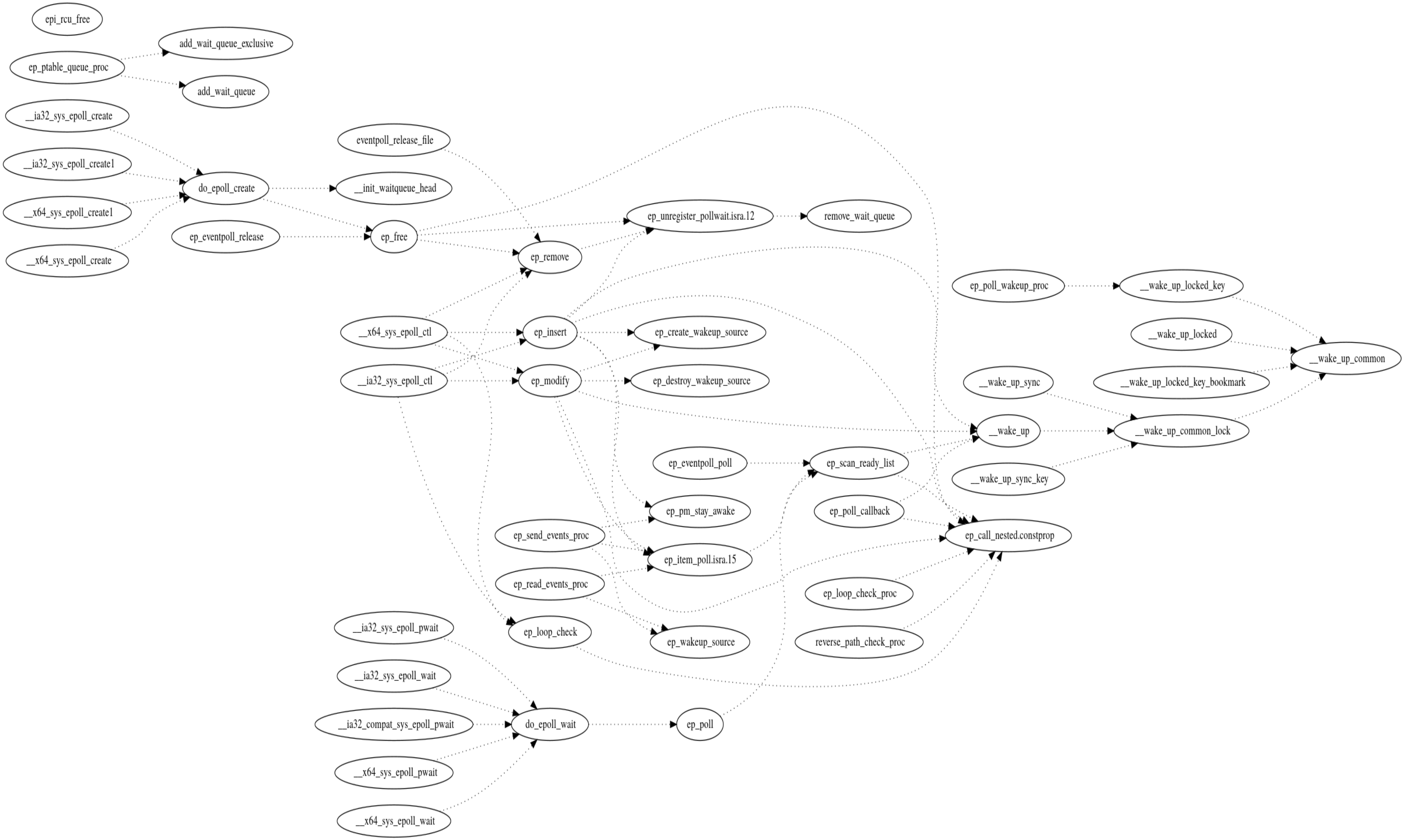
Upstreamed Performance Work

Loosening interrupt safety

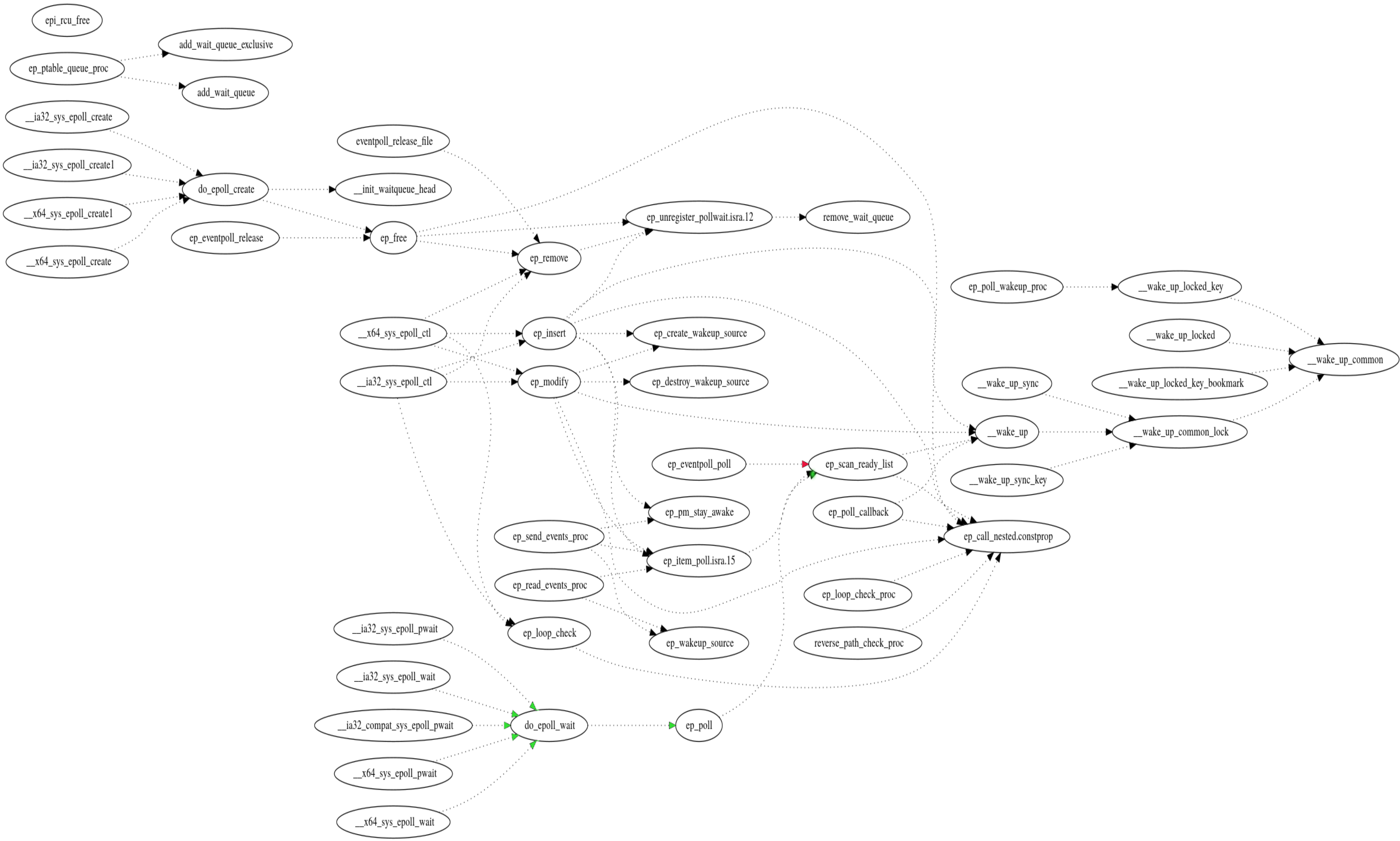
- Epoll is a facility meant for userspace.
  - (Almost) always executes in process context.
  - ep_poll_callback() is often called under irq context.

- Avoid the irq save/restore dance when acquiring ep->lock when we know that interrupts are not already disabled.
  - Benefits in both virtual and baremetal scenarios (ie: x86 replaces PUSHF/POPF for STI/CLI insns).
  - irqsave: needs all flags stable, needs prior insns to retire.
  - irqrestore: changes all flags, expensive insn dependencies.

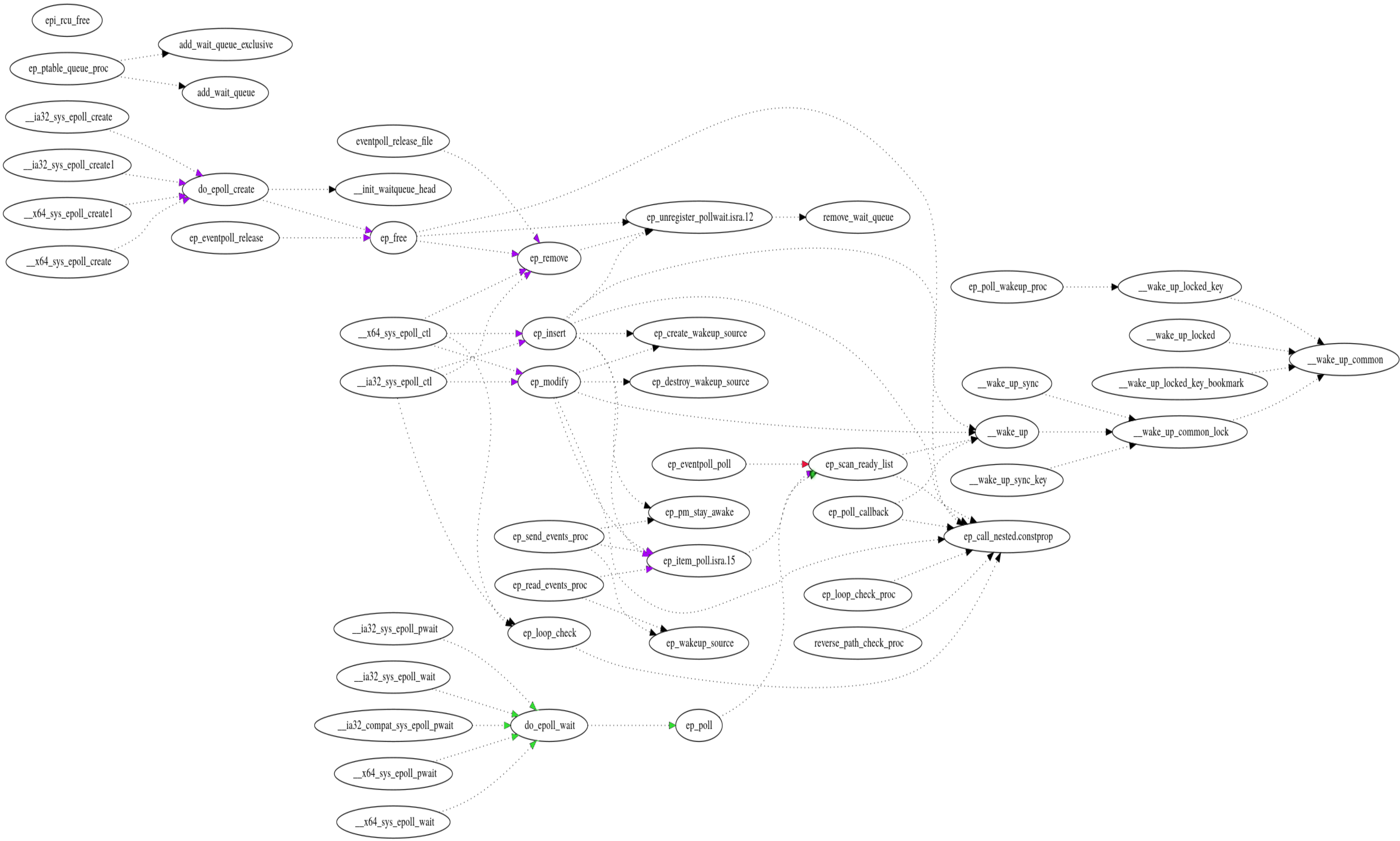
Loosening interrupt safety



Loosening interrupt safety



Loosening interrupt safety



Loosening interrupt safety

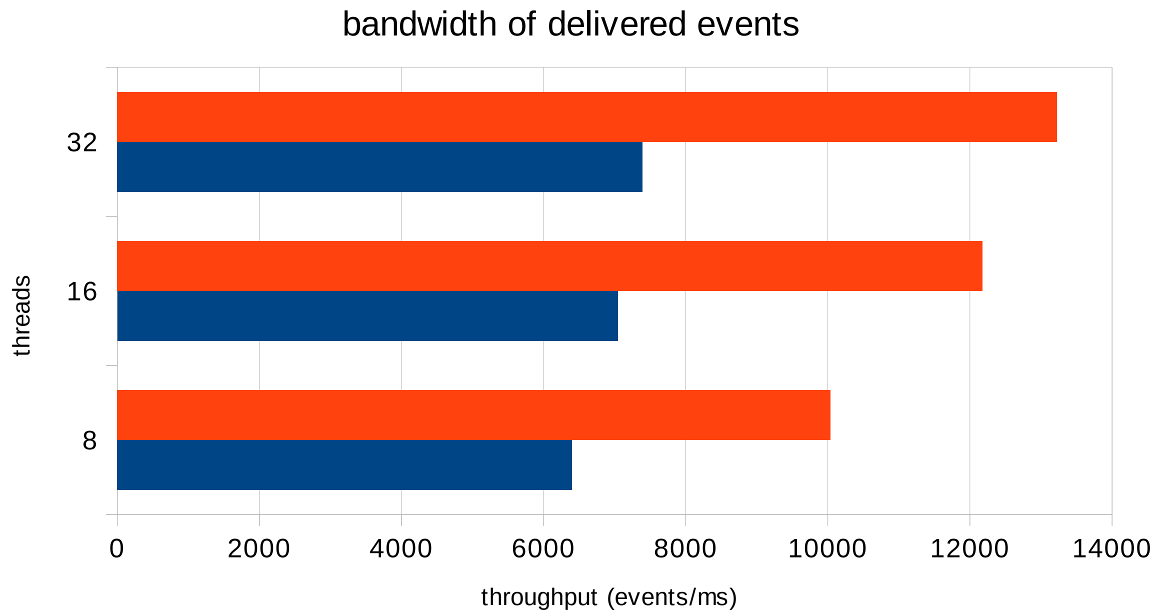
Optimizing ep_poll()

- Main epoll_wait(2) workhorse.
- Locklessly check for available events
  - False positive: we still go into send_events.
  - False negative: we recheck again before blocking.
  - Reduces the scope of the spinlock for the blocking case.
- Do not arm the waitqueue multiple times.
  - Avoid taking locks for every loop iteration (4 lock ops/retry).
- Reduce memory barriers upon failure.

Reduce contention on ep_poll_callback()

- Addresses ep→lock contention.
- Converts ep spinlock to a rwlock.
  - Ready and overflow lists are modified with a read lock + xchg() ops.
  - Stabilize lists elsewhere by using the writer lock.
- Increases the bandwidth of events which can be delivered from sources to the poller.

Reduce contention on ep_poll_callback()



Other Performance Work

Batching interested-list ops

- Epoll doesn’t allow more than one updates on the interest set in a single system call.
  - Avoid multiple system calls.
- Has been proposed upstream 2012, 2015.
- With side channel attacks, is this worth looking at again?
  - Ie: MDS mitigation can flush CPU buffers upon returning to userspace.
- Extend the interface? New syscalls?

Batching interested-list ops

```
int epoll_ctl_batch(int epfd, int flags, int ncmds, struct epoll_ctl_cmd *cmds);
```

- Call atomicity.
  - To succeed do all operations have to succeed?
- Same semantics as non-batched call.

Ring buffer for Epoll

- Fetch new events without calling into the kernel.
  - Ring bufer is shared between the application and the kernel to transmit events as they happen.
- MO is not straightforward.
  - epoll_create2() and EPOLL_USERPOLL
  - epoll_ctl() to add items to the interested-list.
  - mmap() to get at the actual RB.
- Can only be Edge-Triggered.
  - Only one event is added to the RB will be added to the ring buffer when a fd is ready.

SUSE

Ring buffer for Epoll

- Yet another ring buffer in the kernel
  - perf events, ftrace, io_uring, AF_XDP.

- EPOLLEXCLUSIVE is not supported – big drawback.

- API is complex.

Benchmarking Epoll

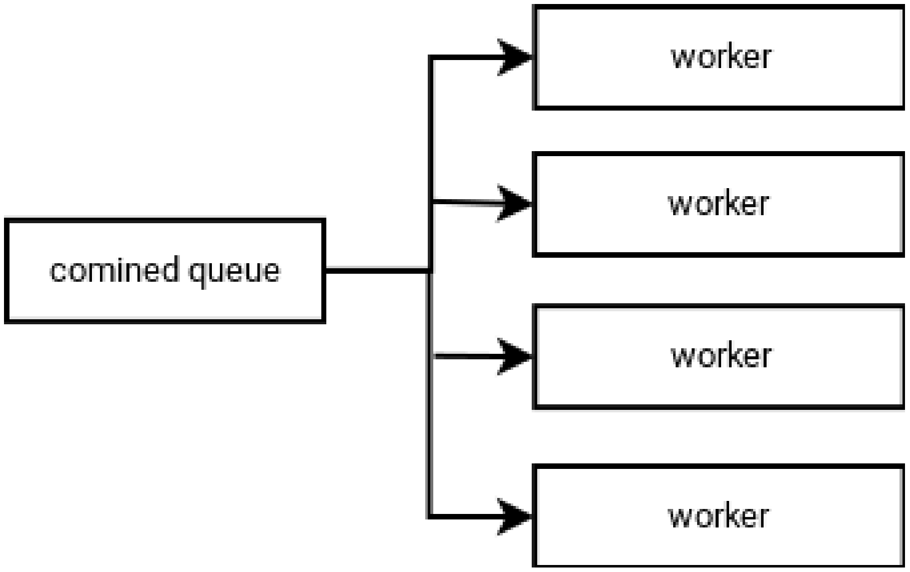
What to benchmark?

- Measure and stress <u>internal</u> changes to epoll.
  - As opposed to comparing against other IO multiplexing techniques.
  - We don’t care about the notification method (socket, eventfd, pipes, etc… they’re all fine).
  - Can be considered pathological – take with a grain of salt; as with benchmarks of any nature.

- Main emphasis on `epoll_wait(2)`.
  - Locking/algorithmic changes.
  - Wakeup latencies.

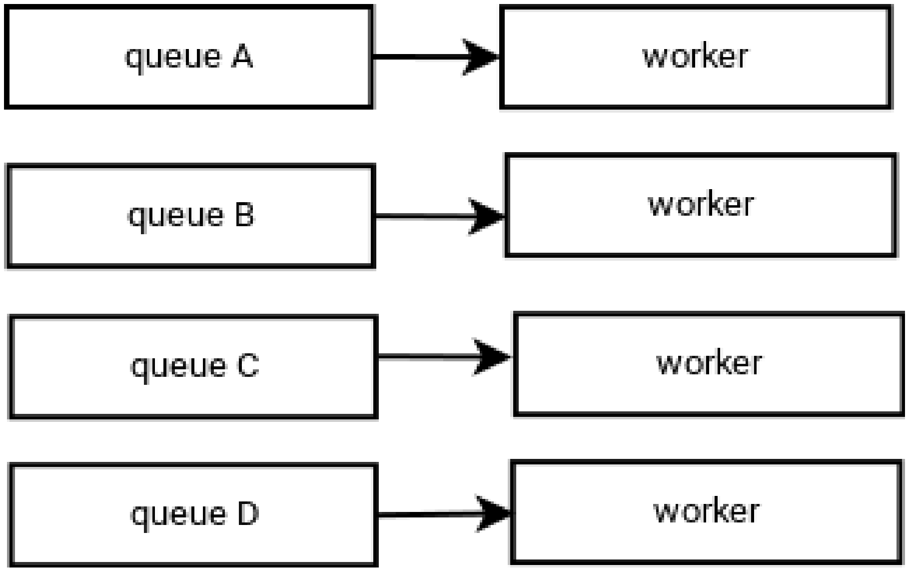
What to benchmark?

- Model (somewhat) common load balancing scenarios.
- Single vs multiqueue (ie: when designing a tcp server)
  - The queue is internal to the kernel via epoll_wait.



What to benchmark?

- Model (somewhat) common load balancing scenarios.
- Single vs multiqueue (ie: when designing a tcp server)
  - The queue is internal to the kernel via epoll_wait.



What to benchmark?

- Shared and private file descriptors.
  - Per ready IO wakeup one and multiple tasks (EPOLLEXCLUSIVE semantics).
- Nested epoll file descriptors.
- Level and Edge-Triggered.

Example: perf-bench (single queue)

```
./perf bench epoll wait -t 16
# Running 'epoll/wait' benchmark:
Run summary [PID 128378]: 16 threads monitoring on 64 file-descriptors for 8 secs.

[thread  0] fdmap: 0x1d87d70 ... 0x1d87e6c [ 36099 ops/sec ]
[thread  1] fdmap: 0x1d87fd0 ... 0x1d880cc [ 36991 ops/sec ]
[thread  2] fdmap: 0x1d88230 ... 0x1d8832c [ 37016 ops/sec ]
[thread  3] fdmap: 0x1d88490 ... 0x1d8858c [ 37158 ops/sec ]
[thread  4] fdmap: 0x1d886f0 ... 0x1d887ec [ 36546 ops/sec ]
[thread  5] fdmap: 0x1d88950 ... 0x1d88a4c [ 36763 ops/sec ]
[thread  6] fdmap: 0x1d88bb0 ... 0x1d88cac [ 36877 ops/sec ]
[thread  7] fdmap: 0x1d88e10 ... 0x1d88f0c [ 36943 ops/sec ]
[thread  8] fdmap: 0x1d89070 ... 0x1d8916c [ 37059 ops/sec ]
[thread  9] fdmap: 0x1d892d0 ... 0x1d893cc [ 37017 ops/sec ]
[thread 10] fdmap: 0x1d89530 ... 0x1d8962c [ 38067 ops/sec ]
[thread 11] fdmap: 0x1d89790 ... 0x1d8988c [ 38082 ops/sec ]
[thread 12] fdmap: 0x1d899f0 ... 0x1d89aec [ 38168 ops/sec ]
[thread 13] fdmap: 0x1d89c50 ... 0x1d89d4c [ 37962 ops/sec ]
[thread 14] fdmap: 0x1d89eb0 ... 0x1d89fac [ 37925 ops/sec ]
[thread 15] fdmap: 0x1d8a110 ... 0x1d8a20c [ 38039 ops/sec ]

Averaged 37294 operations/sec (+- 0.43%), total secs = 8
```

Example: perf-bench (multi-queue)

```
./perf bench epoll wait -t 16 --multiq
# Running 'epoll/wait' benchmark:
Run summary [PID 128415]: 16 threads monitoring on 64 file-descriptors for 8 secs.

[thread  0] fdmap: 0x2c6bd80 ... 0x2c6be7c [ 80864 ops/sec ]
[thread  1] fdmap: 0x2c6bfe0 ... 0x2c6c0dc [ 80864 ops/sec ]
[thread  2] fdmap: 0x2c6c240 ... 0x2c6c33c [ 80864 ops/sec ]
[thread  3] fdmap: 0x2c6c4a0 ... 0x2c6c59c [ 80864 ops/sec ]
[thread  4] fdmap: 0x2c6c700 ... 0x2c6c7fc [ 80864 ops/sec ]
[thread  5] fdmap: 0x2c6c960 ... 0x2c6ca5c [ 80864 ops/sec ]
[thread  6] fdmap: 0x2c6cbc0 ... 0x2c6ccbc [ 80864 ops/sec ]
[thread  7] fdmap: 0x2c6ce20 ... 0x2c6cf1c [ 80864 ops/sec ]
[thread  8] fdmap: 0x2c6d080 ... 0x2c6d17c [ 80864 ops/sec ]
[thread  9] fdmap: 0x2c6d2e0 ... 0x2c6d3dc [ 80864 ops/sec ]
[thread 10] fdmap: 0x2c6d540 ... 0x2c6d63c [ 80864 ops/sec ]
[thread 11] fdmap: 0x2c6d7a0 ... 0x2c6d89c [ 80864 ops/sec ]
[thread 12] fdmap: 0x2c6da00 ... 0x2c6dafc [ 80864 ops/sec ]
[thread 13] fdmap: 0x2c6dc60 ... 0x2c6dd5c [ 80864 ops/sec ]
[thread 14] fdmap: 0x2c6dec0 ... 0x2c6dfbc [ 80864 ops/sec ]
[thread 15] fdmap: 0x2c6e120 ... 0x2c6e21c [ 80861 ops/sec ]

Averaged 80863 operations/sec (+- 0.00%), total secs = 8
```

Thank you.

SUSE®

We adapt. You succeed.