Improving the QEMU Event Loop

Fam Zheng
Red Hat

KVM Forum 2015
Agenda

• The event loops in QEMU

• Challenges
  – Consistency
  – Scalability
  – Correctness
The event loops in QEMU
QEMU from a mile away
Main loop from 10 meters

- The "original" iothread
- Dispatches fd events
  - **aio**: block I/O, ioeventfd
  - **iohandler**: net, nbd, audio, ui, vfio, ...
  - **slirp**: -net user
  - **chardev**: -chardev XXX
- Non-fd services
  - **timers**
  - **bottom halves**
Main loop in front

- **Prepare**
  
  ```c
  slirp_pollfds_fill(gpollfd, &timeout)
  qemu_iohandler_fill(gpollfd)
  timeout = qemu_soonest_timeout(timeout, timer_deadline)
  glib_pollfds_fill(gpollfd, &timeout)
  ```

- **Poll**
  
  ```c
  qemu_poll_ns(gpollfd, timeout)
  ```

- **Dispatch**
  
  - **fd, BH, aio timers**
    
    ```c
    glib_pollfds_poll()
    qemu_iohandler_poll()
    slirp_pollfds_poll()
    ```

  - **main loop timers**
    
    ```c
    qemu_clock_run_all_timers()
    ```
Main loop under the surface - iohandler

• Fill phase
  - Append *fds* in *io_handlers* to *gpollfd*
    • *those registered with qemu_set_fd_handler()*

• Dispatch phase
  - Call *fd_read* callback if (*revents & G_IO_IN*)
  - Call *fd_write* callback if (*revents & G_IO_OUT*)
Main loop under the surface - slirp

- **Fill phase**
  - For each slirp instance ("-netdev user"), append its socket fds if:
    - **TCP** accepting, connecting or connected
    - **UDP** connected
    - **ICMP** connected
  - Calculate timeout for connections

- **Dispatch phase**
  - Check timeouts of each socket connection
  - Process fd events (incoming packets)
  - Send outbound packets
Main loop under the surface - glib

- Fill phase
  - `g_main_context_prepare`
  - `g_main_context_query`
- Dispatch phase
  - `g_main_context_check`
  - `g_main_context_dispatch`
GSource - chardev

- IOWatchPoll
  - Prepare
    - g_io_create_watch or g_source_destroy
    - return FALSE
  - Check
    - FALSE
  - Dispatch
    - abort()

- IOWatchPoll.src
  - Dispatch
    - iwp->fd_read()
GSource - aio context

• Prepare
  - compute timeout for aio timers

• Dispatch
  - BH
  - fd events
  - timers
iothread (dataplane)

Equals to aio context in the main loop GSource...

except that "prepare, poll, check, dispatch" are all wrapped in aio_poll().

```c
while (!iothread->stopping) {
  aio_poll(iothread->ctx, true);
}
```
Nested event loop

- Block layer synchronous calls are implemented with nested aio_poll(). E.g.:

```c
void bdrv_aio_cancel(BlockAIOCB *acb)
{
    qemu_aio_ref(acb);
    bdrv_aio_cancel_async(acb);
    while (acb->refcnt > 1) {
        if (acb->aiocb_info->get_aio_context) {
            aio_poll(acb->aiocb_info->get_aio_context(acb), true);
        } else if (acb->bs) {
            aio_poll(bdrv_get_aio_context(acb->bs), true);
        } else {
            abort();
        }
    }
    qemu_aio_unref(acb);
}
```
A list of block layer sync functions

- bdrv_drain
- bdrv_drain_all
- bdrv_read / bdrv_write
- bdrv_pread / bdrv_pwrite
- bdrv_get_block_status_above
- bdrv_aio_cancel
- bdrv_flush
- bdrv_discard
- bdrv_create
- block_job_cancel_sync
- block_job_complete_sync
Example of nested event loop (drive-backup call stack from gdb):

```c
#0  aio_poll
#1  bdrv_create
#2  bdrv_img_create
#3  qmp_drive_backup
#4  qmp_marshal_input_drive_backup
#5  handle_qmp_command
#6  json_message_process_token
#7  json_lexer_feed_char
#8  json_lexer_feed
#9  json_message_parser_feed
#10 monitor_qmp_read
#11 qemu_chr_be_write
#12 tcp_chr_read
#13 g_main_context_dispatch
#14 glib_pollfds_poll
#15 os_host_main_loop_wait
#16 main_loop_wait
#17 main_loop
#18 main
```
## Challenge #1: consistency

<table>
<thead>
<tr>
<th></th>
<th>main loop</th>
<th>dataplane iothread</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>interfaces</strong></td>
<td>iohandler + slirp + chardev + aio</td>
<td>aio</td>
</tr>
<tr>
<td><strong>enumerating fds</strong></td>
<td>g_main_context_query() + ppoll()</td>
<td>add_pollfd() + ppoll()</td>
</tr>
<tr>
<td><strong>synchronization</strong></td>
<td>BQL + aio_context_acquire(other)</td>
<td>aio_context_acquire(self)</td>
</tr>
<tr>
<td><strong>GSource support</strong></td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Challenges
Challenge #1: consistency

• Why bother?

  - The main loop is a hacky mixture of various stuff.
  
  - Reduce code duplication. (e.g. iohandler vs aio)
  
  - Better performance & scalability!
Challenge #2: scalability

- The loop runs slower as more fds are polled
  - *pollfds_fill() and add_pollfd() take longer.
  - qemu_poll_ns() (ppoll(2)) takes longer.
  - dispatch walking through more nodes takes longer.
Benchmarking virtio-scsi on ramdisk

Disk IOPS degradation with increasing number of fds

- read
- write
- randrw

IOPS of a single disk

Number of scsi disk on the same bus

[Graph showing IOPS degradation with increasing number of 1, 4, 8, 16, 32, 64, 128, 256 scsi disk on the same bus for read, write, and randrw operations.]
virtio-scsi-dataplane

Disk IOPS degradation with increasing number of fds (dataplane)

IOPS of a single disk

Number of scsi disk on the same bus

1 4 8 16 32 64 128 256
Solution: epoll

"epoll is a variant of poll(2) that can be used either as Edge or Level Triggered interface and \textit{scales well to large numbers of watched fds.}"

- epoll_create
- epoll_ctl
  - EPOLL_CTL_ADD
  - EPOLL_CTL_MOD
  - EPOLL_CTL_DEL
- epoll_wait

- \textit{Doesn't fit in current main loop model :(}
Solution: epoll

• Cure: aio interface is similar to epoll!
• Current aio implementation:

  - aio_set_fd_handler(ctx, fd, ...)
  - aio_set_event_notifier(ctx, notifier, ...)

Handlers are tracked by `ctx->aio_handlers`.

- aio_poll(ctx)

Iterate over `ctx->aio_handlers` to build `pollfds[]`. 
Solution: epoll

• New implementation:
  
  - `aio_set_fd_handler(ctx, fd, ...)`
  - `aio_set_event_notifier(ctx, notifier, ...)`

  Call `epoll_ctl(2)` to update epollfd.

  - `aio_poll(ctx)`

  Call `epoll_wait(2)`.

• RFC patches posted to qemu-devel list:
  
Challenge #2½: epoll timeout

• Timeout in epoll is in ms

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

  ```c
  int epoll_wait(int epfd,
                 struct epoll_event *events,
                 int maxevents,
                 int timeout);
  ```

• But nanosecond granularity is required by the timer API!
Solution $2^{1/2}$: epoll timeout

- Timeout precision is kept by combining timerfd:

1. Begin with a timerfd added to epollfd.

2. Update the timerfd before epoll_wait().

3. Do epoll_wait with timeout=-1.
Solution: epoll

- If AIO can use epoll, what about main loop?

- Rebase main loop ingredients on to aio
  - I.e. Resolve challenge #1!
Solution: consistency

• Rebase all other ingredients in main loop onto AIO:
  1. Make iohandler interface consistent with aio interface by dropping fd_read_poll. [done]
  2. Convert slirp to AIO.
  3. Convert iohandler to AIO.
     [PATCH 0/9] slirp: iohandler: Rebase onto aio
  4. Convert chardev GSource to aio or an equivalent interface. [TODO]
Unify with AIO
Next step: Convert main loop to use aio_poll()
Challenge #3: correctness

- Nested aio_poll() may process events when it shouldn't

  E.g. do QMP transaction when guest is busy writing

  1. drive-backup device=d0
     bdrv_img_create("img1")
     -> aio_poll()
  2. guest write to virtio-blk "d1": ioeventfd is readable
  3. drive-backup device=d1
     bdrv_img_create("img2")
     -> aio_poll() /* qmp transaction broken! */

  ...
Solution: aio_client_disable/enable

• Don't use nested aio_poll(), or...
• Exclude ioeventfds in nested aio_poll():

```c
aio_client_disable(ctx, DATAPLANE)
op1->prepare(), op2->prepare(), ...
op1->commit(), op2->commit(), ...
aio_client_enable(ctx, DATAPLANE)
```
Thank you!