Shared Memory Queues
vs
TCP

Which is better for point-to-point and point-to-multi-point intra-processor IPC?

Agenda

- Overview of Point-to-point shared memory queue
- Shared memory queue enhancement to support single sender with multiple receivers
- Observations about the shared memory queue.
- How to support thread pools for sending and receiving
- Simple (but slower) lockless shared memory queue
- Using TCP for IPC
- Performance comparison
  - Point-to-point
  - Point-to-multipoint
  - Impact of batch size on shared memory queue
- Experimental code and spreadsheet containing results can be found in https://github.com/abashandy-github/shm_queue
  - Under ISC license
  - Repository is **NOT public** !!

Shared Memory queue

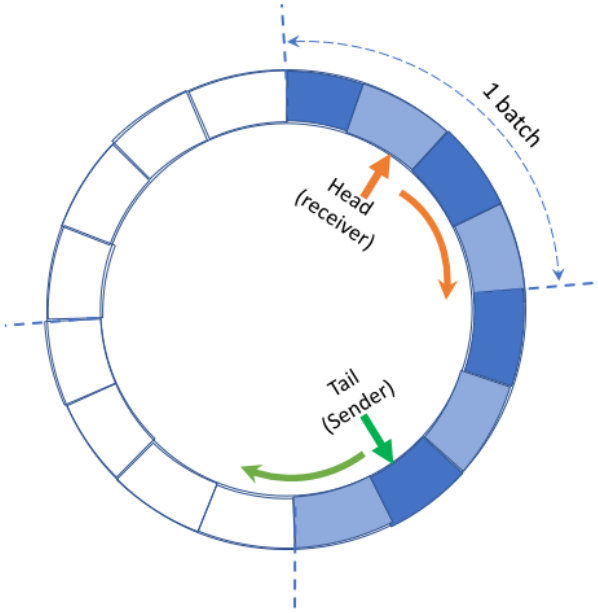
`shm_queue.c`

Objectives

- Shared memory queue for same-processor IPC
- *Reliable*
- Bounded and known-upfront memory usage
  - Includes *backpressure* from receiver(s) to sender
- Support sender and receiver thread-pool
- Supports both point-to-point and point-to-multipoint
- Can be used in conjunction with epoll()/select(),…, etc
  - ➔ hence the sending and receiving thread(s) need not be separate thread(s) (can be one of the applications thread)
- Faster than kernel-based IPC mechanisms (TCP, AF_UNIX, pipe, message queue)

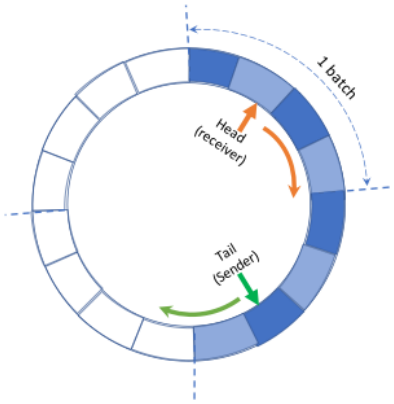
Shared Memory Queue

- Simple Circular list of buffers (each called a “packet”)
- Packets are bundled into *batches*
- *Sender* enqueues at “***tail***” one batch at a time
- *Receiver* dequeues at “***head***” one batch at a time



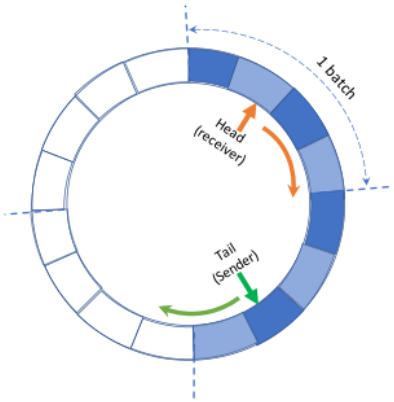
Shared Memory queue: sender

1. Sender locks queue mutex
2. If queue is empty then
   - `wakeup_receiver` = true
3. Calculates a batch size as the min of
   - Preconfigured maximum batch size
   - Number of empty packets available in the queue
   - Packets needed to send all objects that needs to be sent
4. Unlocks queue mutex
5. Populates one batch starting at "tail"
6. Locks the queue mutex
7. Advances "tail" and increments "`num_queued_packets`"
8. If queue is full
   - `high_water_mark_state` = true
9. Unlock queue mutex
10. If (`wakeup_receiver` == true)
    - Pulse receiver
11. If (`high_water_mark_state` == true)
    - Block until pulsed from receiver
12. Goto step 1



Shared Memory queue: receiver

1. `wait_for_sender` = true (this is the initial state because queue is empty at the beginning)
2. If (`wait_for_sender`)
   - Block until pulsed from sender
3. Lock queue mutex
4. Batch is min of
   - Preconfigured maximum batch size
   - Currently queued packets
5. Unlock queue mutex
6. Dequeue a batch starting at "head"
7. Lock queue mutex
   - Advance "head" and decrement "`num_queued_packets`" by batch size
   - If queue is empty
     - `wait_for_sender` = true
   - If (`high_water_mark_state` == true)
     - If (`num_queued_packet` < `low_water_mark`)
       - `high_water_mark_state` == false;
       - `wakeup_sender` = true
8. Unlock queue mutex
9. If (`wakeup_sender` == true)
   - Pulse sender
10. Goto step 2



The "head" is *chasing* the "tail" trying to catch up with it
"Head" is *one batch* behind the "tail" if sender is *active*
"Head" catches up with "tail" if sender is *idle* (nothing to send)

Additional notes

- Blocking is implemented by waiting on a file descriptor using `epoll()*`
- Pulse is implemented by
  - Performing "`write()`" on a file descriptor created by `eventfd()`
    - `eventfd()` file descriptor is created by the sender and transferred to receiver(s) using ancillary SCM_RIGHTS message sent over the AF_UNIX socket **
  - Perform a 1 byte "`write()`" or "`send()`" on the AF_UNIX socket
  - We can als have a TCP socket between the sender and each receiver

* We can also use wrappers such libevent or other system calls such as select()
**Do "man cmsg" to see an into about ancillary messags

Shared Memory Queue: Point to *Multi-point*

- Complications
  - Different receivers consume the same packet at different speeds
  - Different receivers have different batch sizes
  - Some receivers may have consumed all packets and hence will wait for sender's pulse while others are still consuming
  - Some receivers are too slow and hence cause the queue to be full (high_water_mark = true) while others are too fast and cause the queue to be empty
- Basic idea of multipoint enhancement
  - Still have one copy of each packet
  - ➔ one queue and one "tail" pointer for all receivers
  - Maintain different states for different receivers
    - Separate "head" and "num_queued_packets" per receiver
    - Separate "high_water_mark_state" per receiver
    - Separate "wait_for_sender" per receiver

Shared Memory Queue: Point to Multi-point sender modifications

- Sender wakes up receivers as follows
  - Go over all receivers
  - If (receiver[i].wait_for_sender == true)
    - Wakeup receiver[i]
- Sender calculates a Batch size as the minimum of
  - Maximum batch size as configured by user
  - Min Among all receivers of receiver[i].available_packets
  - Packets needed to send all objects that needs to be sent
- Sender queues a batch of packets as usual
- When updating the state of the queue, Go over all receivers
  - Increment receiver[i].num_queued_packets by the batch size
  - If (packets queued for receiver[i].num_queued_packets == queue_len)
    - Receiver[i].high_water_mark_state = true
  - if(receiver[i].wait_for_pulse_from_sender
    - Pulse receiver[i]

Shared Memory Queue: Point to Multi-point Receiver modifications

- Batch size is min of
  - Maximum batch size as configured by user
  - Receiver[i].num_queued_packets
- Receiver dequeues packet as normal
- Receiver updates ***its own*** "head", "num_queued_packets", "wait_for_pulse_from_sender
- If receiver[i].high_water_mark_state = true
  - If receiver[i]. num_queued_packets < low_water_mark
    - If receiver[i] is the LAST receiver to go to high water mark state and then go below low water mark
      - Pulse sender

Observations (1)

- While writing, the sender's "head"
  - At least **part of batch AHEAD** of all receivers' "tail"
  - I.e. while *writing* and *reading* the same batch is always **mutually exclusive**
  - Receiver catches up with sender when sender is idle (queue becomes empty)
  - Sender bumps into receiver with receiver is slow (queue full & high water mark reached)
- Hence dividing a *batch* into *packets* may seem useless
- Instead just treat an entire batch as one packet?
- Dividing a batch into Packets is useful in the following scenarios*
  - The *sender* is *multi-threaded* where each thread takes care of writing and enqueueing one or more packets in the same batch
  - The *receiver* is *multi-threaded* where each thread takes care of reading and dequeueing one or more distinct packet in the same batch
  - It allows different receiver(s) to dequeue at a different rates that other receivers and sender.
    - For example sender's batch can be 20 packets while receiver(s) can dequeue a batch at 2,3,4 packets

*There may be other scenarios that I cannot think of

Observation (2)

- What is the use of having a queue that is many batches long?
- Can't we just divide the queue into two batches?
  - One batch being read by receiver(s)
  - The other batch being written to by sender
- A long queue acts as a cushioning buffer that allows greater speed discrepancy between sender and receiver
  - I.e. it increases the ability of the sender to batch when receiver(s) are temporarily slow
  - Allows receiver(s) to be closer to the sender and hence reduces the delay
    - I.e. the smaller a batch size the smaller the delay from sending till receiving

Observation (3)

- The biggest impact on speed is number of system calls
- We noticed performance degradation when
  - the number of times a sender has to block until it receives a pulse from receiver (due to queue becoming full then going below low water mark) OR
  - Number of times a receiver has to block until it receives a pulse from sender (due to queue becoming empty)
- If these two numbers increase, then the overall time to transmit the same amount of data also increases.

Observation (4)

- Being a contiguous block of memory, there is significant leverage to CPU data cache even in the multicore scenario
  - In theory, the only cache miss occurs when the head or tail makes one full rotation around the circular queue
- There is an inherent ***backpressure*** because once the queue is full the sender blocks until the receiver(s) dequeue enough packets
  - So there is no buffer overflow or loss
- Most importantly: The implementation is relatively simple

Thread pools for with shared memory queue*

* I did not do a prototype for the idea in this section

Objective

- Allow sender to use multiple threads to queue packets
- Allow receiver(s) to use multiple threads to dequeue packets
- Still maintain other shared memory queue objectives

Idea (1)

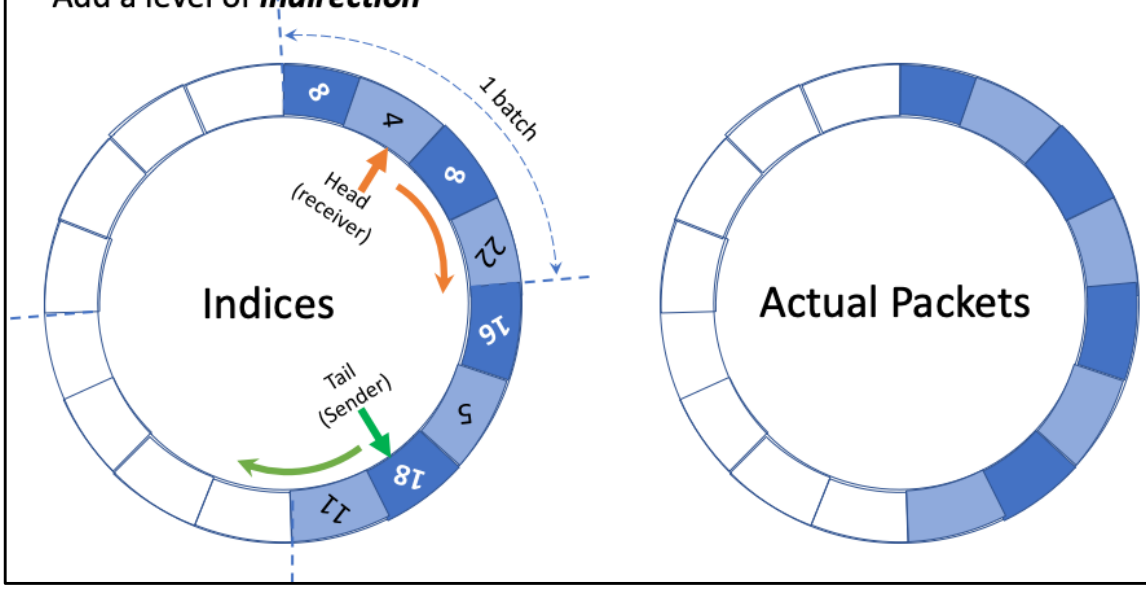
- Assume max batch consists on “n” packets
- Assume the sender is using “n” threads
- Assume receiver is using “m” thread and receiver mac batch
- One threads in the sender and receiver is designated as *master* thread
- Sender
  - Thread “i” grabs packet “i” in the batch
  - Thread “i” populates packet “i” in the batch
  - Once all threads populate their respective packets, master thread performs the same actions done in the single thread case
- Receiver
  - Thread “i” grabs packet “i”
  - Thread “i” processes the data in packet “i” in the batch
  - Once all threads process data their respective packets, master thread performs the same actions done in the single thread case

Idea (1): complications

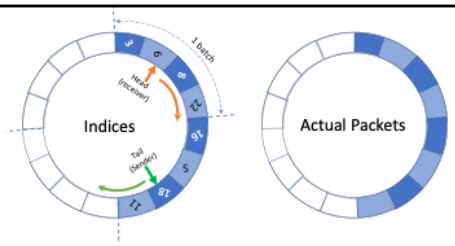
- Signaling has to occur for multiple threads instead of single thread
  - If we can fix the problem that we saw with single `write()` to and `eventfd()` file descriptor can wakeup multiple processes waiting on that file descriptor, then multiple thread signaling can be optimized (there is still an overhead due to multi-threading)
- Synching the threads at the sender and receiver
  - Multiple threads are populating/consuming sent/received packets by the head and tail of the queue are updated once
  - We do not want to update the "head" and "tails" of the queue before all threads have populated/consumed the all packets to be sent/received in the batch
  - We can use a ***barrier*** to make sure that all threads have populated/consumed all packets in a batch before updating the queue
  - Other ideas are most welcomed

Idea (2)

- Add a level of ***indirection***



Idea (2)



- Thread "i" grabs packet "i" in the batch
- When it is done, it puts the index in the first slot available in the "indices" circular buffer
- "head" and "tail" rotate on the "indices" circular buffer instead of the packets
- The only advantage
  - The "tail" and "head" can be advanced *without waiting* for all threads to populate all packets in a batch
  - It would be great to point other advantages and/or enhance this idea
- Cons:
  - Populating the index in the "indices" circular buffer" has to happen one thread at time
    - ➔ mutex to protect the "indices" circular buffer"
  - Additional memory access because we have to dereference the packet index in the "indices" circular buffer to the packets circular buffer
  - Now that packets are accessed ***out of order***, the leverage of data cache is *reduced*
  - Still signaling has to occur for multiple thread

Serialized Lockless Shared memory*

`Shm_simple_queue.c` (**Lockless**)

* Main objective is performance comparison

Objective

- Very simple shared memory queue
- Sending and receiving are serialized

Idea

- Queue consists of one packets
- Receiver connects to sender
  - Receives eventfd() file descriptor via SCM_RIGHTS ancillary message
  - waits on the eventfd() file descriptor using epoll_wait()
- Sender
  - fills up the packet
  - signals receiver throughwrite() to the eventfd() file descriptor
  - Waits on the eventfd file descriptor for a signal from the receiver
- Receiver
  - Gets unblocked and empties the packet
  - Signals the sender by write() to the eventfd() file descriptor
  - Waits on the eventfd file descriptor for a signal from the sender

Using TCP for IPC*

`tcp_queue.c` (Simplest TCP-based IPC)

* The primary objective of this implementation is to compare TCP with shm_queue.c

Objective

- Make the simplest and fastest TCP-based IPC
- We want to use to measure performance against shared memory queue

Idea: sender

- Establish TCP session between sender and receiver(s)
- Sender outer loop: loop over number of objects to be sent (e.g. 1B objects, each size 50)
  - Inner Loop: Loop over number of objects that can fit one buffer
    - Increment counter in object
    - Copy object to buffer
    - Advance pointer in the buffer
  - Exit inner loop
  - Sending loop: Loop over all receivers
    - Send the same buffer to each receiver using “send()”
  - Exit sending loop
- Go back to the beginning of the inner loop

Receiver

- Establish TCP session with sender
- Receiver outer Loop
  - Receiver same size buffer as sender using recv()
  - Inner loop: Loop over all objects in that buffer
    - Copy each object to an outside buffer
    - Validate that the counter is sequential
    - Advance the pointer to the next object in the buffer
    - Continue until all objects have been read
  - End inner loop
- Go back to the beginning of outer receiver loop

Performance Comparison

Objective

- Compare point-to-point IPCs
- Compare point-to-multipoint shared memory queue to TCP
- Explore the effect of batch size on shared memory queue performance

Environment in which experiments where conducted

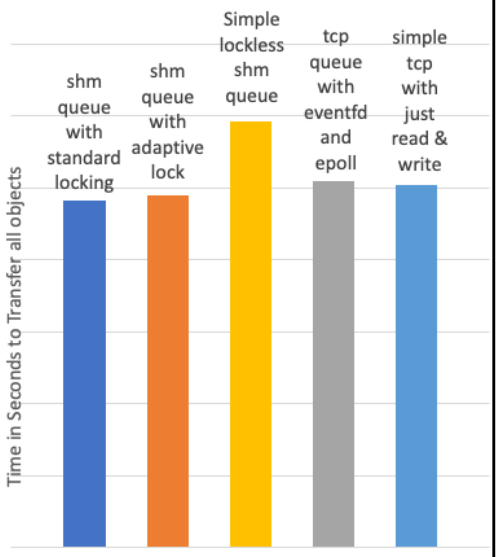
- I used a VM running VERSION="18.04.2 LTS (Bionic Beaver)", kernel version 4.18.0.
- The VM was assigned 4 cores on an X86 processor with 8 cores
- Total number of objects to be transferred: 1,000,000,000
- Size of each object: 50 bytes
- Performance was measured by the time to transfer all objects
  - I do "`clock_gettime(CLOCK_MONOTONIC, &start_ts);`" at the beginning of sending/receiving *, **
  - I do "`clock_gettime(CLOCK_MONOTONIC, &end_ts);`" after finishing sending/receiving all 1B objects
  - Subtract to get the difference

* For shm_queue.c, the sender sends a sync pulse so that all receivers take s time stamp at the same tim

** for "tcp_queue.c()" I measure the time reported by the last receiver connected to sender because other receivers will have extra time because the sender will not start sending unless all reecivers connet to sender

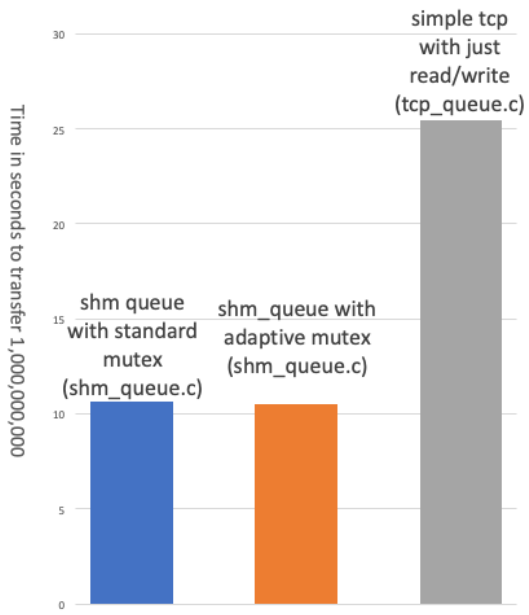
Point-to-Point IPC comparison

- For this test, we made tried to keep a batch at the size of 100,000 bytes
- Remember that with every batch, the sender and receiver exchange signalling and/or adjust the queue size
- Shm_queue is about 6-7*% better* than the simplest TCP
- We did not expect a huge difference between shm_queue and TCP
  - TCP has been around for more than 30 years
- Adaptive mutex is 1% better than standard mutex
  - But the variance is largest



Point-to-Multipoint

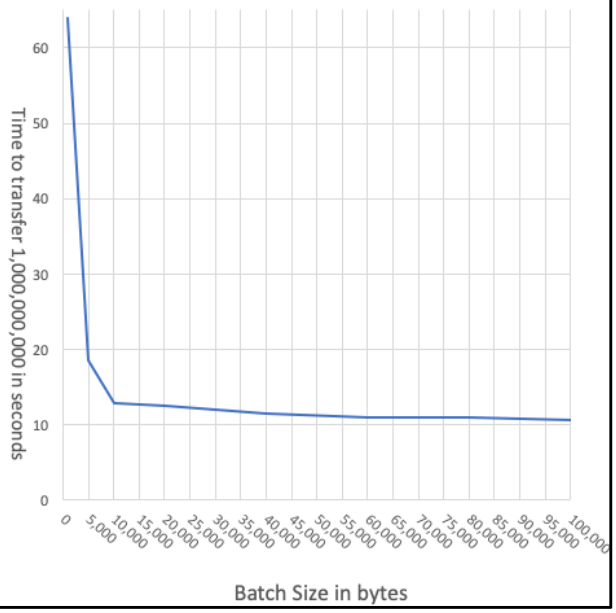
- 1 sender and 3 receivers*
- We only compared `shm_queue.c` with the simplest TCP (`tcp_queue.c`)
- Shm_queue is ***significantly better***
  - about 2.5 times faster
- Why not 3 times faster for 3 receivers
  - Still has to signal 3 receivers
  - Contention between sender and 3 receivers
  - Contention between receivers
  - TCP signaling is probably faster because all of it is done in the kernel
- Why TCP is slow?
  - TCP has to
    - Send each batch 3 times
    - Maintain 3 distinct TCP sessions



* Did that because I had a total of 4 cores assigned to the VM on which I ran the tests

Effect of Batch size on shared memory queue

- Remember that at every batch both sender and receiver(s)
  - Calculate the maximum batch size
  - Update queue parameters (e.g. “tail” and “head”)
  - May signal the receiver(s) and sender, respectively
- Hence the smaller the batch
  - the more frequent the queue is updated and hence
  - more system calls and mutex lock/unlock as well as
  - higher probability of mutex contention among the sender and receiver(s)
- Larger than a certain size, enlarging the batch size yields **very little** improvement
- Below a certain size, reducing the batch size ***significantly reduces*** performance*
- Results expected due increase in number of *system calls* and *signaling* compared number of objects to send
  - E.g. perform queue maintenance every 500 objects instead of every 50,000



* Still things are **not linear**. For example, comparing the largest batch size 100,000 with the smallest 1,000, one of them is 100 times larger. However the performance drop is only 6.5 times.