Inter-process links for Snabb

Max Rottenkolber <max@mr.gy>

Wednesday, 31 October 2018

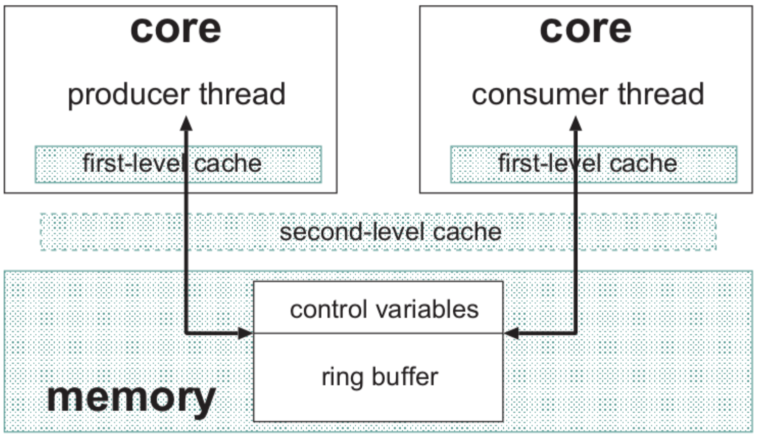
Snabb (https://github.com/snabbco/snabb) allows your network functions to utilize multiple cores of your CPU, which is inevitable in the age of increasing core counts. Applications do this by forking into additional processes where each process is typically bound to a dedicated core. Together they form a *process group*, and while each process executes an independent event loop, some memory segments are shared across the group.

With hardware support for receive-side scaling (RSS), processes can share a network I/O device while receiving and transmitting on dedicated hardware queues, but they can also forward packets from one to another. The latter is accomplished through low-overhead *inter-process links* (http://snabbco.github.io/tag/v2018.09.html#inter-process-links-apps.interlink.), which will be the topic of this article.

"Systems programmers are the high priests of a low cult"—Robert S. Barton

Fast, multi-core, SPSC queues

At the *heart* (https://github.com/snabbco/snabb/blob/v2018.09/src/lib/interlink.lua) of Snabb's inter-process links lies a single-producer/single-consumer (SPSC) queue that is optimized for low overhead, multi-core operation. For these inter-process queues we have adapted a ring buffer design described in *A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring* (Patrick P. C. Lee, Tian Bu, Girish Chandranmenon), called *MCRingBuffer*.



Multi-core architecture as illustrated by the *MCRingBuffer paper* (http://www.cse.cuhk.edu.hk/%7Epclee/www/pubs/ipdps10.pdf).

The crux of the matter is that, due to the cache hierarchy of modern multi-core architectures, one core can invalidate another core's first-level *cache lines* (https://en.wikipedia.org/wiki/CPU_cache#CACHE-LINES). That is, when one core writes to a memory location that is cached by another core then that cache entry is invalidated. When the second core then tries to read that memory location its value has to be fetched from the second-level cache or, even worse, main memory. Because the second-level cache and main memory have much higher latency than the first-level cache, we want to avoid this on the fast path as much as possible.

```
struct link {
  int read, write;
  struct packet *packets[LINK_RING_SIZE];
}
```

Basic *ring buffer* (http://en.wikipedia.org/wiki/Circular_buffer) used in Snabb.

On x86_64 processors, the basic ring buffer above can be inherently consistent during multi-core operation when used as a single-producer/single-consumer queue: the `read` and `write` cursors are mutated by the producer and consumer exclusively, and to store a doubleword[1] is an atomic operation. Total store ordering (TSO) takes care of the rest.

There is only one problem: assuming the producer and consumer run on separate cores, whenever the consumer increments the `read` cursor it will also invalidate the producer's cached value of the cursor, and vice versa for the `write` cursor. If the producer and consumer increment the cursors for each packet they enqueue or dequeue then we might end up with a whole lot of accesses to higher-latency memory for no reason. This phenomenon is called *false sharing* (https://en.wikipedia.org/wiki/False_sharing).

```
struct interlink {
  int read, write;
  char pad1[CACHELINE-2*sizeof(int)];
  int lwrite, nread;
  char pad2[CACHELINE-2*sizeof(int)];
  int lread, nwrite;
  char pad3[CACHELINE-2*sizeof(int)];
  struct packet *packets[LINK_RING_SIZE];
} __attribute__((packed, aligned(CACHELINE)))
```

Ring buffer with eliminated false sharing.

The *MCRingBuffer* introduces producer and consumer local copies of the `read` and `write` cursors, and segregates the memory layout so that the local cursors do not share a cache line with shared memory locations. The effect is that the producer and consumer can mutate their local cursors without invalidating each other's caches. We can then amortize the latency of mutating the shared cursors by enqueuing and dequeuing packets in batches, and updating them only after each batch.

- 1. On x86_64 a doubleword is 32-bit wide. Because Snabb exclusively supports x86_64 processors we can rely on the fact that an `int` is represented as a doubleword.

The life cycle of a queue

Any Snabb process can create queues, and attach to or detach from any queue at any time. To make sure that no more than a pair of producers and receivers attach to any queue at a given time, and queues are deallocated when they are no longer used… well, enter the next synchronization problem.

Who	**Change**	**Why**
any	*none* → `FREE`	A process creates the queue (initial state).
consumer	`FREE` → `RXUP`	Consumer attaches to free queue.
consumer	`TXUP` → `DXUP`	Consumer attaches to queue with ready transmitter.
consumer	`DXUP` → `TXUP`	Consumer detaches from queue.
consumer	`RXUP` → `DOWN`	Consumer deallocates queue.
producer	`FREE` → `TXUP`	Producer attaches to free queue.
producer	`RXUP` → `DXUP`	Producer attaches to queue with ready receiver.
producer	`DXUP` → `RXUP`	Producer detaches from queue.
producer	`TXUP` → `DOWN`	Producer deallocates queue.

Valid state transitions in the life-cycle of a queue.

The life cycle of a queue is modeled as a finite-state machine defined by the state transitions listed above. Notably, the state machine only blocks when trying to attach to a queue that already has both a producer and a consumer attached. Regular attaching and detaching are both lock-free: producers and receivers do not need to wait on each other before they start enqueuing and dequeuing packets.

```
local cas_t = "bool (*) (int *, int, int)"
local function cas (Dst)
   | mov eax, arg2
   | lock; cmpxchg [arg1], arg3 -- compare-and-swap; sets ZF flag on success
   | mov eax, 0                 -- clear eax for return value
   | setz al                    -- set eax to 1 (true) if ZF is set
   | ret
end
```

`cas(dst, old, new) -> true|false` Atomic compare-and-swap; compare `old` with value pointed to by `dst`. If equal, stores `new` at `dst` and returns *true*. Else, returns *false*.

To synchronize state transitions between processes we have added low-level *synchronization primitives* (https://github.com/snabbco/snabb/blob/v2018.09/src/core/sync.dasl) for x86_64 using the *DynASM* code generator. The atomic *compare-and-swap* (https://en.wikipedia.org/wiki/Compare-and-swap) (CAS) routine used to transition between states of the queue life cycle is shown above.

Borrowers and lenders

You may have noticed that we are passing mere pointers to packets through our inter-process links as opposed to copying the packet between processes. So how does that work?

```
function rebalance_freelists ()
   if group_fl and freelist_nfree(packets_fl) > packets_allocated then
      freelist_lock(group_fl)
      while freelist_nfree(packets_fl) > packets_allocated
      and not freelist_full(group_fl) do
         freelist_add(group_fl, freelist_remove(packets_fl))
      end
      freelist_unlock(group_fl)
   end
end
```

Redundant packets are released into the group's shared freelist.

Snabb processes within the same process group will map *DMA* (https://en.wikipedia.org/wiki/Direct_memory_access) pages that we use to store packets into their own address space on demand via a

SIGSEGV handler (https://github.com/snabbco/snabb/blob/v2018.09/src/core/memory.c). This means that processes can freely pass packet pointers around, and it will "just work." Still, each Snabb process maintains its own packet *freelist* (https://github.com/snabbco/snabb/blob/v2018.09/src/core/packet.lua#L48) to which it allocates and frees packets. If the producer keeps allocating packets and the consumer keeps freeing them the producer's freelist will drain, and the consumer's freelist will overflow.

```
function allocate ()
   if freelist_nfree(packets_fl) == 0 then
      if group_fl then
         freelist_lock(group_fl)
         while freelist_nfree(group_fl) > 0
         and freelist_nfree(packets_fl) < packets_allocated do
            freelist_add(packets_fl, freelist_remove(group_fl))
         end
         freelist_unlock(group_fl)
      end
      if freelist_nfree(packets_fl) == 0 then
         preallocate_step()
      end
   end
   return freelist_remove(packets_fl)
end
```

Packet allocation consults the process group's shared freelist before asking the operating system for more memory.

To avoid this from happening, the process group maintains a shared freelist into which processes can release their extraneous packets, and from which they can reclaim their missing packets. When a process allocates packets it first tries to take them off its internal freelist, then it consults the group's freelist, and only then, as a last resort, it will allocate new packets from the operating system.

```
if counter.read(breaths) % 100 == 0 then
   packet.rebalance_freelists()
end
```

Rebalance the freelists every hundred breaths.

While the *spinlock* (https://github.com/snabbco/snabb/blob/v2018.09/src/core/sync.dasl#L29-L52) that processes use to synchronize on the shared freelist is fast, rebalancing packets is certainly not free. Hence, processes release extraneous packets only in fixed, spaced intervals, and only try to access the shared freelist once they run out of packets. In practice, this leads processes to allocate enough packets to sustain their workload during these intervals.

Acknowledgments

I feel like I need to credit R. Matthew Emerson for *researching and coming up* (https://github.com/snabbco/snabb/pull/772) with most of the design for Snabb's inter-process links, as well as Luke Gorrie whose code I studied in order to arrive at finite-state machine that governs queue life cycle sychronization. Finally, I want to express my gratitude towards *NLnet* (https://nlnet.nl/) for funding part of my work on inter-process links.