An MP-capable network stack for DragonFlyBSD with minimal use of locks

Aggelos Economopoulos

1 Introduction

When DragonFlyBSD forked from FreeBSD 4, it inherited a decades-old network stack. While it is well tested and a reference implementation, the code was a bad fit for modern multicore hardware. This situation was only going to get worse as the number of cores on typical processor chips is expected to increase over time. The main problem was the assumption that, except for hardware interrupts, "since there is only one CPU, only one thread can access the network stack data structures". As for other kernel subsystems, the usual way to ensure this assumption remained true on an SMP system has been to force mutual exclusion between processes attempting to run in kernel mode by requiring them to obtain a lock (the Big Giant Lock or BGL) first.

Of course, this "solution" doesn't allow the kernel to use more than one CPU at a time. At this point, the traditional approach is to try and partition the problem. This approach, (commonly know as fine grained locking) protects selected (sets of) significant data structures using appropriate locking primitives. This way, the kernel can run on all system CPUs, as long as no two threads try to access the same data structure. However, some data structures are more frequently accessed and this results in contention for the corresponding locks. Heavy lock contention can significantly affect the system's ability to linearly scale to large numbers of CPUs, which is the ultimate, if unattainable, goal. Again, the traditional solution is to split the lock into more locks, each protecting a part of the original data structure. Following this path, the FreeBSD and Linux kernels have now got to thousands of locks in a running kernel. While fine-grained locking demonstrably scales well to tens of CPUs, one should keep in mind that most of this locking is pure overhead on uniprocessors and small-scale SMP systems where concurrency is inherently limited.

Another issue that becomes apparent with large lock numbers is that it is no longer easy to ensure the correctness of the locking operations. Even experienced kernel programmers may find it hard to adhere to the multitude of rules that define the appropriate locking order to avoid deadlock scenarios. It is certainly not a trivial task to deduce what went wrong when such a problem occurs. While straightforward at first, fine-grained locking quickly becomes cumbersome as the programmers try to make the system scale on more CPUs.

2 Organization of the DragonFly network stack

One of the goals set when DragonFly was first created was to try and minimize the complexity associated with locking. To this end, the project developers moved most of the protocol code into separate threads that communicate with the rest of the kernel using messages. At this point, a short description of the DragonFly messaging model is in order.

2.1 Message passing semantics

The basic abstraction we need to know about is the message port. DragonFly defines a few types of ports, but the network code only uses thread-owned ports. Access to such a port is only allowed by code running on the same CPU as the thread that the port belongs to. Such code can just enter a critical section and manipulate the port directly (e.g. linking a message in the port's message queue). A thread running on a different CPU that wants to queue a message must send an IPI to the CPU of the owning thread in order to access the port. Needless to say, IPIs are not fast, so it is essential to batch IPI delivery. Sending an IPI while the target CPU is processing its IPI queue will not generate an actual hardware interrupt in DragonFly and the network code takes special care to minimize the number of IPIs necessary. As can be seen in the table, the basic DragonFly kernel API for message passing is quite simple.

lwkt_sendmsg
Send asynchronous message
lwkt_domsg
Send synchronous message
lwkt_forwardmsg
Forward message to some other port
lwkt_getport
Retrieve next queued message without blocking
lwkt_waitport
Block waiting for a particular message

2.2 Protocol threads

In DragonFly, the code responsible for TCP and UDP packet processing runs in separate kernel threads. There is one TCP and one UDP thread for each system CPU. Packets belonging to the same TCP connection are always handled on the CPU of the TCP thread handling the connection. Any TCP timers associated with a TCP connection are guaranteed to run on this same CPU. This ensures effortless mutual exclusion and cache locality. Each protocol thread has an associated message port. Protocols other than TCP and UDP are handled by generic per-CPU network threads.

2.3 Data flow

When a network interface (this discussion focuses on ethernet interfaces without loss of generality) receives a packet, the driver interrupt routine stores the data in a newly allocated mbuf. The interrupt code (which runs in the context of a dedicated interrupt thread) decides which protocol thread to send the mbuf to based on the packet type and content. E.g. for TCP, this requires looking at the source and destination address and port numbers. The code takes special care to link together mbufs destined for

the same CPU and use IPIs to do the actual message delivery on the target CPUs so that the number of IPIs is significantly reduced.

Mbufs embed the actual message structure and the code listening on the destination message port can easily get to the enclosing mbuf. After this, the protocol thread can process the incoming mbuf as usual. In the common case of a valid data packet, the data is delivered to the receive sockbuf of the corresponding socket and any userspace threads waiting for input are notified.

A userspace thread that wants to send some data either passes them over to the protocol thread which then queues the data in the send sockbuf or (the new behaviour) directly queues the data in the sockbuf and asynchronously notifies the protocol thread that more outgoing data is available. The data will be transmitted when the protocol thread gets around to servicing the message (or earlier, if there already was pending data in the sockbuf; if the protocol thread starts sending data because of some earlier message, it will send all the available data). Transmission takes place by placing the data in the interface output queue and arranging for the driver if_start() routine to run. This procedure runs in the context of a separate ifnet thread. There are as many ifnet threads as there are system CPUs and, for a given interface, if_start runs on the same CPUs as the interrupt thread for this interface's interrupt handler.

The reader should keep in mind that this description does not mention a number of complications because of IP fragment reassembly, IP multicast handling, ALTQ, the packet filtering code and several other factors that a general purpose implementation must concern itself with. We also omit some optimizations that would confuse rather than aid in the understanding of the network stack organization.

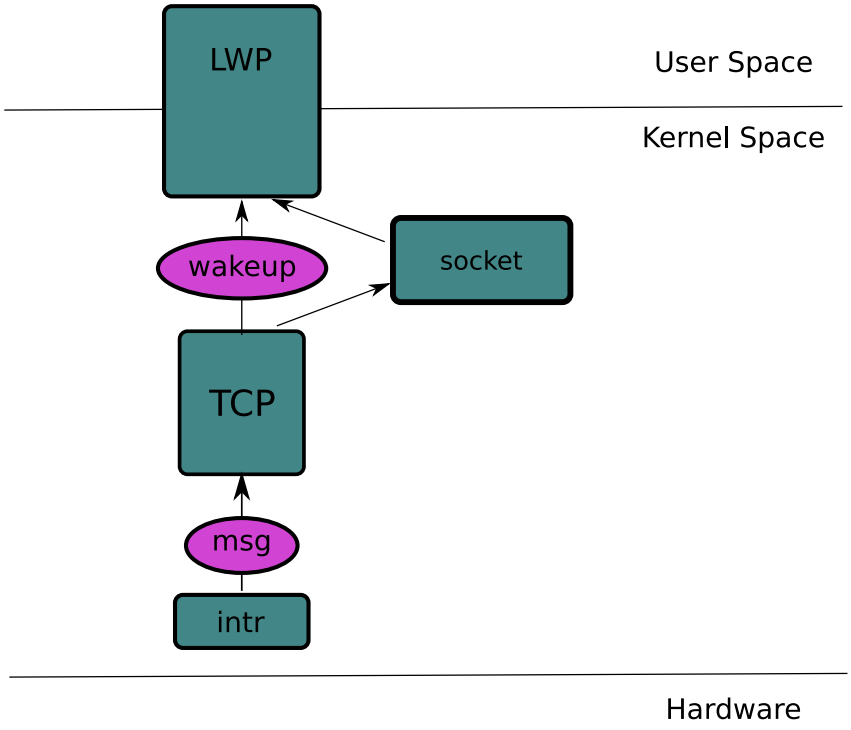
3 Issues

The work to make the network code MP-safe begun with identifying the data structures that are accessed both by userland threads running in kernel mode and the pure-kernel protocol threads. The most interesting of those is the socket buffer or sockbuf.

3.1 The socket buffer

Each socket has a send sockbuf that is written to by userspace threads and that the corresponding protocol thread reads from, as well as a receive sockbuf that is only written to by a single protocol thread but can be read by any userspace thread that has a reference to the socket. In other words, we have both a multiple producers / single consumer and a single producer / multiple consumers problem to solve. Solutions to such problems exist, but are either too complex or incur high overhead or require special instructions that may not be available on all architectures of interest. Notice, however, that the side of the protocol thread is always the "single" side. Therefore, the two cases can be turned to a, comparatively simpler, single producer / single consumer (abbreviated SPSC) problem by using a lock to synchronise accesses by the userspace threads. There are obvious lockless and wait-free algorithms for the SPSC case. The main advantage of this "shortcut" is that, in contrast to using a socket buffer lock, a protocol thread need never wait for userspace to be done with the sockbuf. On top of that, further work might make it possible to eliminate the need for the user-side lock when there is only one reference to the socket, thus achieving completely lock-free operation. The next three section will discuss the implementations we considered, including the one (cupholders) which we ended up adopting.

Figure 1: Receive path



3.1.1 The ring buffer

There are several ways to tackle the SPSC scenario; a popular one and the one initially pursued was to use a ring buffer. The naive implementation uses a read index that is incremented by the consumer every time it reads an entry from the buffer and a write index that the consumer updates after writing a new entry. This is adequate from a correctness perspective, but requires the producer to access the read index (to know whether the buffer is full) and (more importantly, for reasons that will be explained later) the consumer to read the write index in order to detect buffer underflow. In other words, there is continuous cacheline thrashing. A better and equally simple approach is the one suggested in FastForward [1], where the reader and writer can detect underflow/overflow just by reading the buffer entries. As long as there is more than a cacheline's worth of entries in the buffer, there will not be any cacheline bouncing between the producer and consumer CPUs (assuming they are different, otherwise there is no issue). Our original implementation was FastForward-style. Unfortunately, the network code uses the sockbuf character count (i.e. the amount of bytes in the buffer) to decide whether there is data to receive or if there is enough free space to store incoming data. It is possible to change this behaviour and look at the sockbuf entries directly, but there were other problems that made use of the ring buffer unattractive.

Another issue was the space wasted on unused buffer entries. Since the buffer was fix-sized, it had to be large enough so it wouldn't overflow even when the TCP window reached its maximum size. That meant a 4KB buffer, or 8KB of kernel memory reserved for every socket (for the send and receive sockbufs), even if no data was going through it. Naturally, an inefficient peer could always fill the buffer by sending small packets because it is no longer trivial to do mbuf coalescing. In that case, we would be forced to silently drop some in-window data. Punishing an inefficient peer may or may not be acceptible in all usage scenarios.

3.1.2 M_CORAL

The next approach we tried was to keep a singly-linked list and make sure that the consumer never removes the tail. This way, the producer doesn't need to worry about the mbuf going away while it is linking in a new mbuf. This required adding a new mbuf flag (`M_CORAL`) to mark an mbuf whose data has been consumed but it is kept around because it is the last one in the sockbuf. The consumer can use this flag to eventually remove the mbuf as soon as there is a new list tail. While valid and efficient as a strategy, the implementation was more complex than the ring buffer code. More importantly, mbufs are allocated from different memory pools depending on whether there's an associated mbuf cluster. If the mbuf has been allocated together with an mbuf cluster, they must be deallocated together. If we need to keep the cluster around as well, we are again wasting memory (although not as much memory, and not for sockets that have never been used), which is what we set out to avoid.

3.1.3 Cupholders

At this point it was suggested to add another level of indirection, which led to the birth of the cupholder structure. A cupholder merely contains a pointer to the next (if any) cupholder and a pointer to an mbuf. We only link the cupholders in the sockbuf and make sure there is always at least

one cupholder present. With careful use of memory barriers, we can make sure the producer and consumer can safely run without any synchronization. Cupholders may solve the excessive memory usage issue, but they come with their own disadvantages. They increase the cache footprint of the sockbuf-related data and they must be dynamically allocated. Having to go through the objcache allocator makes a measurable difference in microbenchmarks (although the effect is barely noticeable for practical tests). In the future, we will probably revisit the M_CORAL approach in order to resolve the memory deallocation issues. One could try to make the ring buffer resizable as well, only this would quickly invalidate the ring buffer's primary advantage which is very low code and algorithm complexity. Exploration of different approaches is easy as the APIs are more or less compatible between the different implementations.

Regardless of the approach, a lockless sockbuf comes with the interesting complication that we can no longer provide a "stable" character count (recall that the character count is the number of bytes in the sockbuf). It turns out that this is not a big problem. We can synthesize an approximate character count and the value we return is a lower bound on the amount of data available if we are inquired by the consumer and an upper bound if inquired by the producer. So if the character count informs the consumer that there is available data, that data cannot go away (since there can only be one consumer at a time). Similarly, if the producer is told there is space in the sockbuf, that space will remain available since no one else can add data to the sockbuf. Our sb_reader API ensures that a routine accessing the sockbuf will get a stable character count while it is running.

The other issue is avoiding the lost wakeup problem. Some process context code can try to read data from the sockbuf, find out that there isn't any and go to sleep expecting a wakeup from the protocol code when new data arrive. But if the wakeup comes after the check for available data and before the process goes to sleep, it will be lost and the process might remain blocked forever. This issue is resolved by simply messaging the protocol thread requesting a reply as soon as an appropriate predicate (that there is more data than a specified value or the the connection state has changed) becomes true. Subsequently, the process blocks waiting for the reply message. If more data arrived after our check, the protocol thread will reply immediately and we can continue.

3.2 The protocol control block

The network stack keeps protocol-specific data in a separate structure that is linked to from the socket (there is an 1:1 correspondence). This structure is called the protocol control block or PCB. There are different types of PCBs. All internet family protocols maintain an "inpcb" that keeps track of the local (and maybe the foreign, for connected sockets) address and port numbers associated with the socket and a variety of IP options and flags. For TCP sockets, the inpcb also points to a TCP control block (also known as "tcpcb") which contains the TCP-specific protocol data (window information etc.).

The inpcbs (and therefore the tcpcbs) are kept in a per-cpu hash table. We use the same hash function as for assigning a TCP packet to a thread. This way, the TCP threads will only do a lookup on the CPU-local part of the hash table. As would be expected, accesses to the tcpcb originate mostly

from the TCP threads; the only exception was querying and setting TCP options which happened from process context. The obvious fix was to message the appropriate TCP thread instead.

While UDP doesn't need its own PCB (it just makes use of the inpcb), its PCB issues are far more interesting than TCP's. The original code runs under the MP lock so it can simply use a global hash table. Unfortunately, breaking up the hash table in per-CPU tables is not as straightforward as we could wish for.

The problem is what to use as input to the hash function (which would also be used for assigning packets to the UDP threads). The inpcb for a UDP socket may or may not have a foreign address and port set, depending on whether the user has called connect() on the socket or not. On top of that, a connect() can change these fields at any time. The local address can change from under us as well. If we were to use these fields to calculate the CPU number, we would have to move the inpcb between CPUs when any one of the them changed (assuming we weren't lucky enough that the new CPU was the same as the old one). This shouldn't be too hard to handle by marking the old inpcb as "in deletion" and only deleting it after we have installed a copy of it on the new CPU. For now, we just hash on the local port, but an important issue is that, e.g. for a DNS server, neither the remote address and port nor the local address are normally set. In that case, all DNS packets will all go through only one CPU. This cripples the scalability of our DNS server.

Another approach would be to replicate inpcbs with wildcard entries on all CPUs. Since UDP does not guarantee in-order data delivery, we could have one sockbuf per CPU as well. In that case the process side would pull data in, say, a round-robin fashion. It is uncertain how well existing applications would handle a data stream that is routinely out-of-order though.

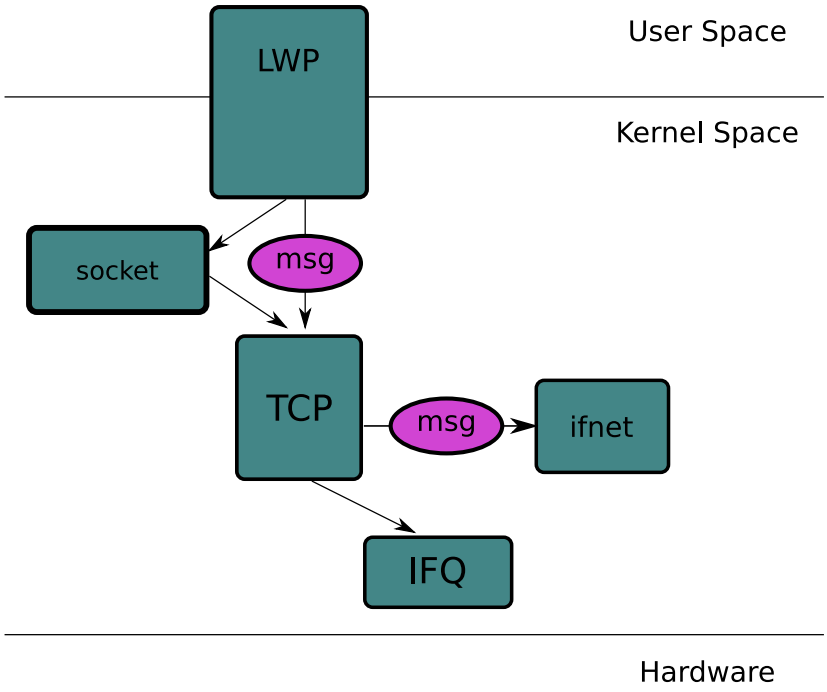
3.3 The socket

The last major structure that is accessed by both process context and protocol thread code is the actual socket. This structure serves a number of different purposes, therefore its fields must be treated separately. To begin with, there are fields such as the socket type and the pointer to the protocol switch (a structure consisting of function pointers) that are set early in the lifetime of a socket and remain fixed thereafter. Then, there are fields such as the socket options that are only modified from process context but, as long as updates happen atomically, it is acceptable to race. If, for example, two user threads try to update a socket option, without any synchronization of their own, they cannot predict the final value of the option anyway, so we might as well just allow the race. Of course, while they do not modify the socket options, the protocol threads use their values, so we needed to go through the protocol code to make sure it does not depend on the option values remaining constant while it is running.

Listening sockets also need to keep track of sockets representing in-progress or not yet accepted connections. For this purpose, these sockets are linked into doubly-linked lists and a socket might be removed from the lists at any time, regardless of its position in the list. For once, we went with the spinlock approach. A queue lock in the listening socket controls manipulation of the linked lists and the sockets contained therein. We can reevaluate our approach if this turns out to be a bottleneck in the future.

Some fields, such as the upcall and

Figure 2: Send path



system call emulator fields are only accessed by kernel subsystems that still run under the BGL so they didn't require any special attention. For others, like SIGIO information, we just added a new message type so the operation can be performed in protocol thread context.

Finally, we needed to deal with the socket state transitions. The state field encodes not only the connection status but also whether the socket can send or receive more data and, for connections that haven't been accepted yet, which listening socket queue they're in. Most of these flags are not especially problematic (it is all right if they change asynchronously) with the exception of flags like SS_CANTRCVMORE. The process side code checks for this flag before blocking waiting for data, but the protocol thread can set it after the check and just before the process goes to sleep. Since we now block by sending a message, even if we race the protocol thread will notice that we are waiting for data on a socket that can't receive any and will wake us up by replying to our message. It goes without saying that in all of the cases above the socket state flags are updated using atomic operations.

3.4 Additional work

Along with making the accesses to shared data structures multiprocessor-safe we needed to change some of the message semantics. For instance, a process sending data would send a synchronous message containing an mbuf pointer to the protocol thread, which would then queue the mbuf in the "send" sockbuf and perform the actual sending. This is obviously suboptimal. The new code will queue the

data in the sockbuf directly from process context and issue an asynchronous "notify" message to the appropriate thread port. The sockbuf API allows the consumer (in this case a protocol thread) to recognise "new" data as opposed to data that has been transmitted and awaits acknowledgement. For now, this strategy is only used for "plain" TCP data (i.e. not for UDP or TCP out-of-band data).

The other interesting optimization is that we added the option not to immediately run a higher-priority thread (such as a protocol thread) when sending it a message. The default behaviour is to forcibly switch to that thread, but that is inappropriate when there is a lot of data to send. The new behaviour is to set a special message flag so that the protocol thread will be scheduled but the sender can keep running and continue adding new data. Eventually, the protocol thread will run and process all queued data en masse.

4 Performance measurements

This is just the beginning of the optimization work. What is needed is measurements of our current performance in a number of different workloads that can serve as a reference for comparison. Getting these measurements is a work in progress. The main issue is that the mainline kernel can already saturate a couple of gigabit links, so it is not straightforward to demonstrate scalability to large numbers of CPUs. Trivial tests have been encouraging however. Hopefully, DragonFly will have a 10Gbit driver before too long and we will be able to fix any remaining performance issues and publish results of real-world testing.

5 Conclusions

Removing the requirement for the BGL from the socket layer and the TCP and UDP threads was a task that was long overdue. That task is not yet complet; we need to fix a few (known) remaining issues and do a lot of real-world testing to uncover the bugs that are bound to exist. Just as importantly, we need to produce performance data that will justify our approach and indicate that one need not use ever more fine grained locking too achieve MP scalability.

6 Acknowledgements

The author would like to thank Sepherosa Ziehau, Matthew Dillon, Simon Schubert and Kornilios Kourtis for many discussions on the problems encountered. Special thanks to Matthew Dillon for contributing both code and ideas when real life interfered with the planned schedule. This project doubles as the author's diploma thesis in the National Technical University of Athens ECE department.

References

[1] John Giacomoni, Tipp Moseley, and Manish Vachharajani: FastForward for Efficient Pipeline Parallelism