Reducing Lock Contention in a Multi-core system

by: Randall Stewart

Why do we lock?

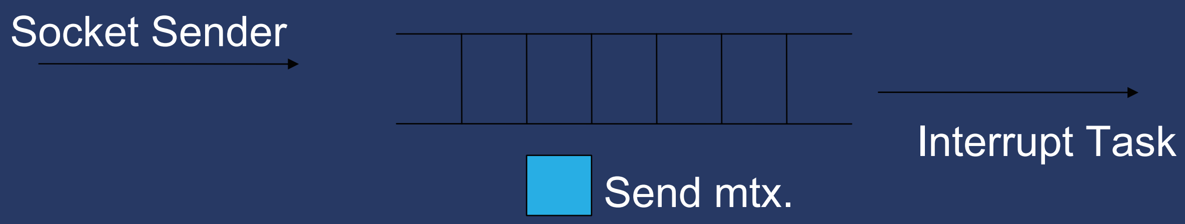
- With the advent of today's multi-core CPU's more and more operating systems are moving to an Symmetric Multi-Processor (SMP) environment!
- Each operating system must thus find ways to ensure sane and coherent operation when multiple CPU's access the same data structures simultaneously.
- One of the most common coherency mechanisms is the mutex.

How do we lock in the FreeBSD kernel?

- Mutex's provide a simple gate that allows one CPU to access a data structure while another waits its turn.
- In user land pthread_mutex's are available for this purpose, but not so in the kernel.
- In the FreeBSD kernel we have the “mtx” structure.
- Like pthread_mutex's these structures, once initialized, can be locked and unlocked for exclusive access to a data structure.

An example used by the SCTP stack in FreeBSD and MAC OS/X.

- In the SCTP implementation we have quite effectively used locking to allow a sender of data (the socket api user) and the transmitter of data (the interface interrupt task) to share a data structure via a mutex.



With a simple scheme

- With this simple scheme we solve the locking problem and our two or more CPU's do not have a problem.
- However we gain one down-side, lock-contention.
- Lock contention is how often one thread holds the lock while the other has it.
- The less lock contention, the more parallel our process will be and thus we will better utilize the multi-core systems we have available.

So how can we measure lock contention?

- FreeBSD comes with a kernel level tool-kit for this very purpose.
- If we build our kernel with the "LOCK_PROFILING" option we can measure our lock contention. (man LOCK_PROFILING)
- Our config file looks like:

```
....
options LOCK_PROFILING
...
```

- Build the kernel in the usual way and reboot

Getting a “lock profile” run

- Lock profiling on your new kernel is NOT enabled by default.
- You can turn on/off/examine lock profiling by:

```
 sysctl -a | grep lock | grep prof | grep debug
debug.lock.prof.stats: No locking recorded
debug.lock.prof.collisions: 0
debug.lock.prof.hashsize: 4096
debug.lock.prof.rejected: 0
debug.lock.prof.maxrecords: 4096
debug.lock.prof.records: 0
debug.lock.prof.acquisitions: 0
debug.lock.prof.enable: 0
```

Getting a “lock profile” run

- Change the enable flag to 1.

```
sysctl -w “debug.lock.prof.enable=1”
```

- Now run any tests that you want to profile.
- After you are done change the sysctl back to '0'
- Now do a sysctl -a > my_file.txt
- Vi/emacs your file and scan down until you find the symbol debug.lock.stats
- You should see a header/numbers that look like

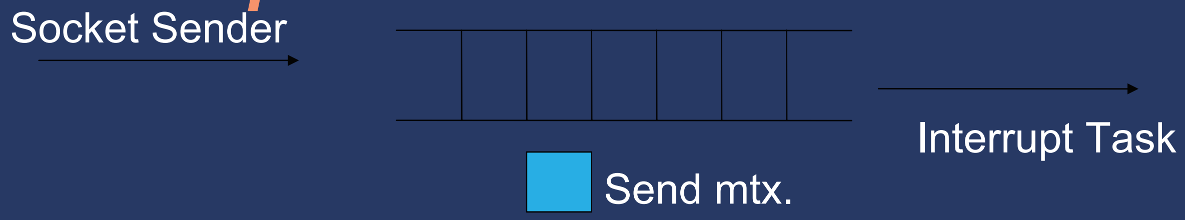
```
max   total wait_total avg  wait_avg cnt_hold cnt_lock name
```

- And lots of numbers.

Mutex Profiling results

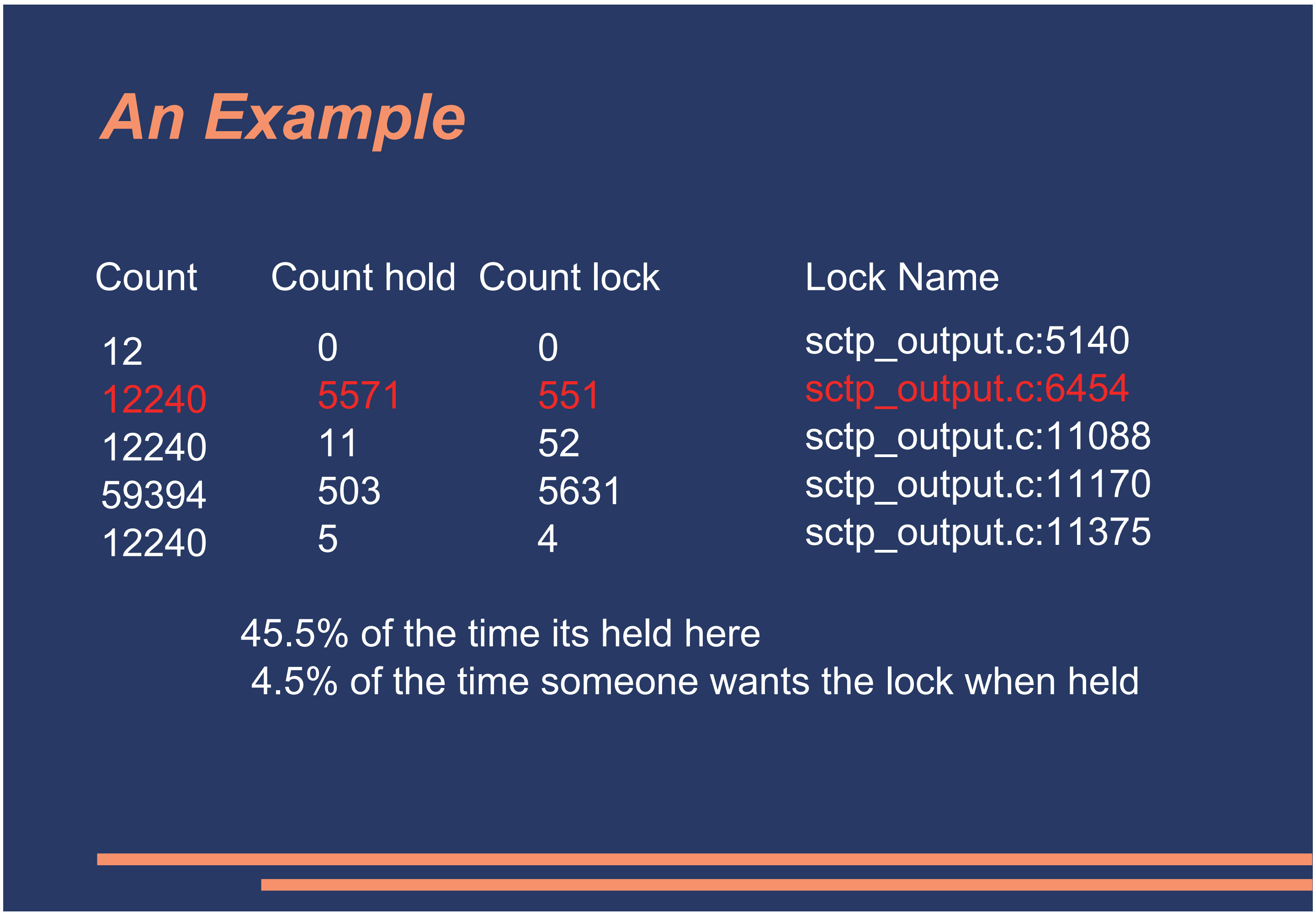
- Max – the max time this point waited in microseconds.
- Total – the total hold time in microseconds.
- Wait_total – the total accumulated wait time.
- Count – the number of times this lock was at this point **
- Avg – The average hold time in microseconds.
- Wait_Avg – the average wait time in microseconds.
- Cnt_hold – The number of times this lock was held when someone else wanted it. **
- Cnt_lock = The number of times someone else held the lock at this point **
- Lock name – the lock name and file and line number.**

An Example



Count	Count hold	Count lock	Lock Name
12	0	0	sctp_output.c:5140
12240	5571	551	sctp_output.c:6454
12240	11	52	sctp_output.c:11088
59394	503	5631	sctp_output.c:11170
12240	5	4	sctp_output.c:11375

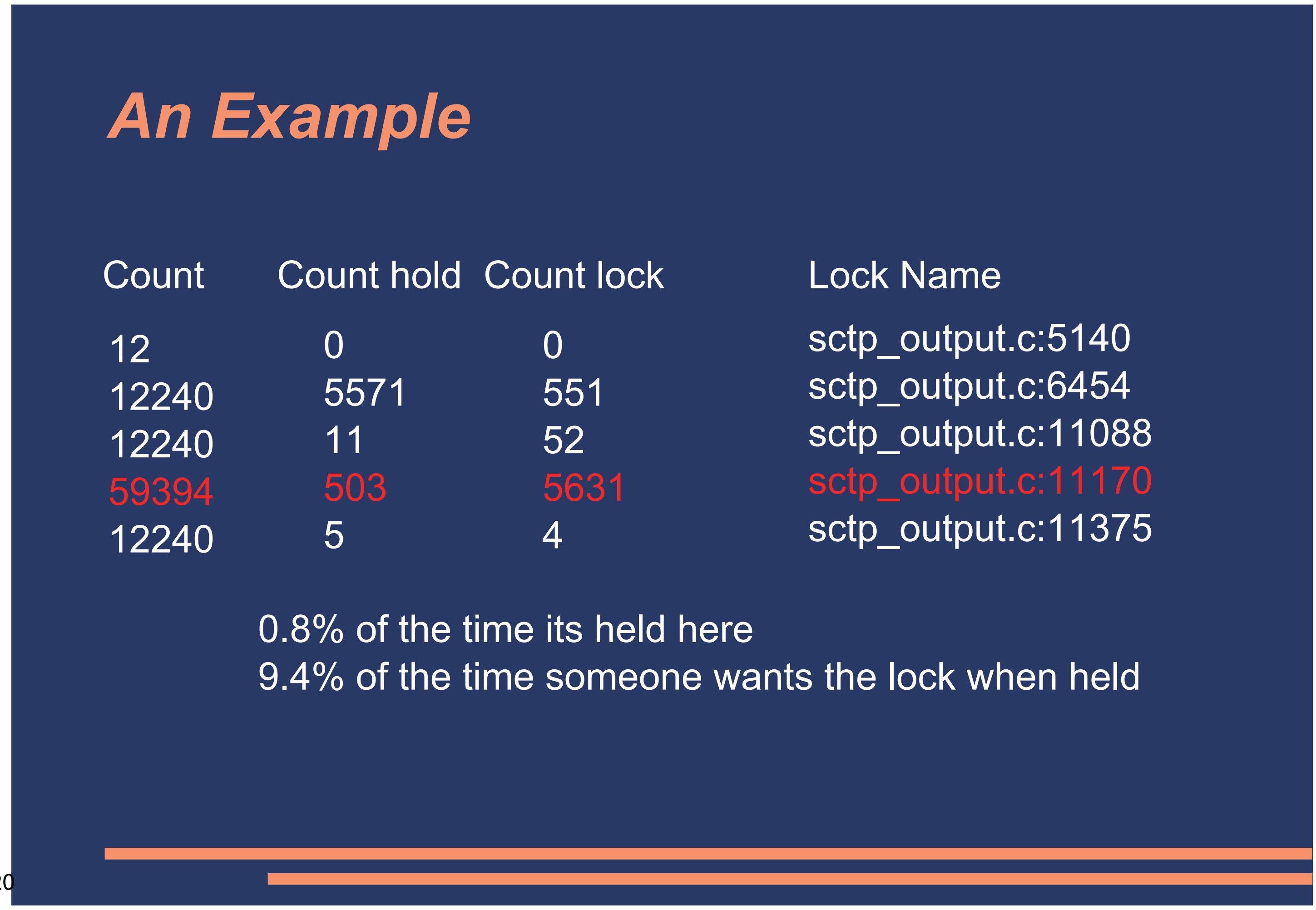
Socket sender
Interrupt transmitter



An Example

Count	Count hold	Count lock	Lock Name
12	0	0	sctp_output.c:5140
12240	5571	551	sctp_output.c:6454
12240	11	52	sctp_output.c:11088
59394	503	5631	sctp_output.c:11170
12240	5	4	sctp_output.c:11375

45.5% of the time its held here
4.5% of the time someone wants the lock when held

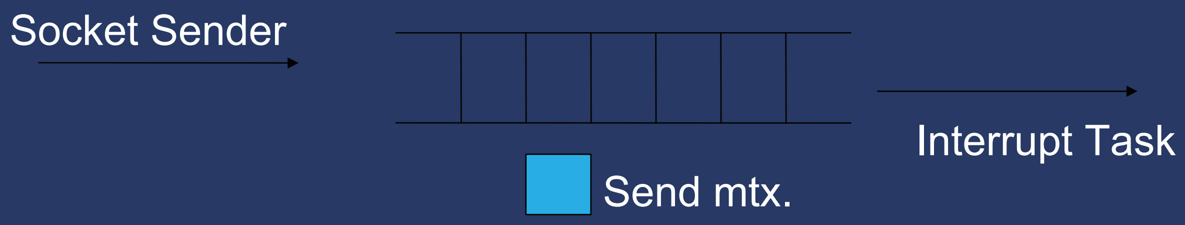


An Example

Count	Count hold	Count lock	Lock Name
12	0	0	sctp_output.c:5140
12240	5571	551	sctp_output.c:6454
12240	11	52	sctp_output.c:11088
59394	503	5631	sctp_output.c:11170
12240	5	4	sctp_output.c:11375

0.8% of the time its held here
9.4% of the time someone wants the lock when held

So how can we reduce contention and preserve sanity?



Is really

```
struct name {
        struct type *tqh_first; /* first element */
        struct type **tqh_last;       /* addr of last next element */
};
```

Where

```
struct name {
        struct type *tqh_first; /* first element */
        struct type **tqh_last;       /* addr of last next element */
};
```

- The socket sender always appends to the tail
- The interrupt transmitter always pulls from the head and may not always pull the whole message off (considering that a msg can be larger than the PMTU).
- So can we reduce locking?

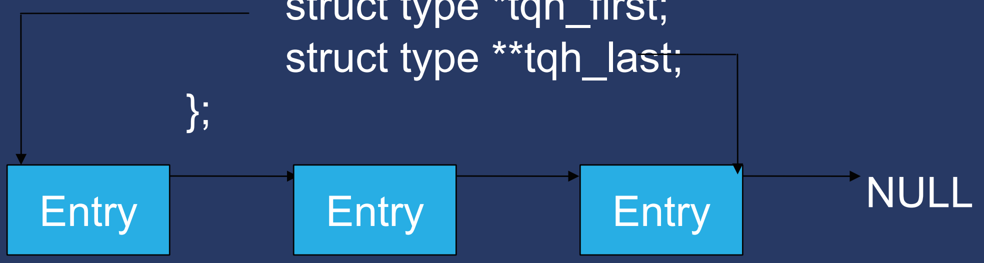
We can observer about the sender

```
struct name {
        struct type *tqh_first; /* first element */
        struct type **tqh_last;       /* addr of last next element */
};
```

- The sender never knows if the transmitter is active and is never sure if the list is empty or has entry's on it (tqh_last points to the head when empty).
- When adding data, we only use tqh_last.
- But without foreknowledge the socket sender MUST always lock the structure.

We can observer about the transmitter

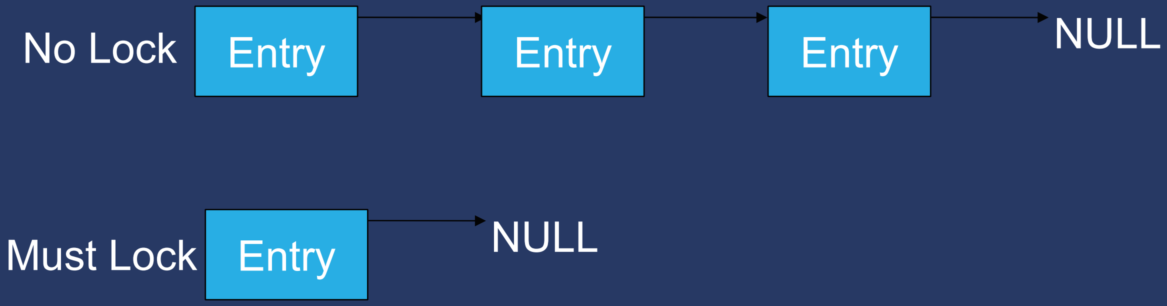
```
struct name {
        struct type *tqh_first;
        struct type **tqh_last;
};
```



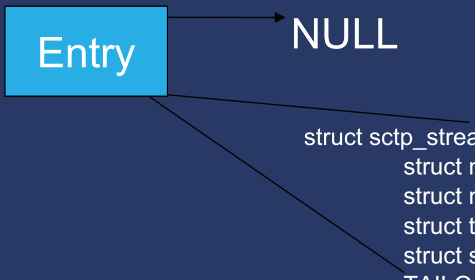
- But what about the transmitter?
- It does know:
  - If it is going to pull the entry off.
  - It can tell if there is already a next entry on the queue.

We can observer about the transmitter when Pulling an entry.

- So from this knowledge could we:
  - Only lock if we will pull the entry from the queue?
  - Don't lock if there is a next item in the queue (since the other thread is inserting there)?



So what about contention on the same entry, when more data is added?



```
struct sctp_stream_queue_pending {
        struct mbuf *data;
        struct mbuf *tail_mbuf;
        struct timeval ts;
        struct sctp_nets *net;
        TAILQ_ENTRY (sctp_stream_queue_pending ) next;
        uint32_t length;
```

- The data being added by the socket sender is a chain of mbufs.

So what about contention on the same entry, when more is added?

```
struct sctp_stream_queue_pending {
        struct mbuf *data;
        struct mbuf *tail_mbuf;
        struct timeval ts;
        struct sctp_nets *net;
        TAILQ_ENTRY (sctp_stream_queue_pending ) next;
        uint32_t length;
```

- If we always update the size last, after appending, then the transmitter will always see either the correct size, or a reduced size.
- Since we use mcopym() with the size, this limits us so when contending for one being added the most that can happen is we will take less than we could have.
- Note that we use atomic_add_int() to assure a barrier and that the compiler does not give us a surprise.

So what two things is the transmitter doing?

- When it goes to add data, don't get a lock unless the size of the copy will exhaust the mbuf completely. This allows continued addition to a message without the transmitter locking.
- When the transmitter decides to remove an entry it will only lock if the “next” pointer is NULL.

Results after our modification

Count	Count hold	Count lock	Lock Name
288	64	4	sctp_output.c:5141
344	157	16	sctp_output.c:6517
33549	19	232	sctp_output.c:11151
45787	1	0	sctp_output.c:11437

Note that the redesigned algorithms have one less lock.

Socket sender

Interrupt transmitter

The results show

- A huge drop in the percentage that the socket sender contends with the transmitter from 45% to .02%
- Very rarely does the transmitter even get a lock, its still a high percentage of contention but with a drop from 12,252 lock requests to 632.

Conclusion

- When adding a shared resource to a SMP O/S one needs to:
  - Carefully consider your data structures that the various locks protect.
  - Examine the level of lock contention.
  - Try to craft mechanisms that allow one of the threads/cpus to NOT lock when possible.
- No two problems are the same but the base concept presented here can be applied to both kernel and user level code.

Other things that can be done in Kernel land (use with caution)

- When wanting to cache resources for quick re-use per CPU lists can be created.
- One can use the critical_enter/critical_exit call to prevent being scheduled.
- The code would look something like:

```
free_item(entry_t *item) {
  critical_enter();
   cpu = curcpu;
   LIST_INSERT_HEAD(&cache[cpu].list,
                                       item, next);
  critical_exit(); }
```