Kernel concurrency

- Open-source FreeBSD operating-system kernel
- Large: millions of lines of code
- Complex: thousands of subsystems, drivers, ...
- Extremely concurrent: supports 128+ HW threads
- Netapp, EMC, Panasas, Dell, Apple, Juniper, Cisco, McAfee, Netflix, Verio NY Internet, Yahoo!, Verisign, ...
- Used at CL (Capsicum, CHERI, TESLA, SOAAP, ...)
- Employs many of the principles we have talked about
Brief history

- 1980s DARPA-funded Berkeley Standard Distribution (BSD)
- UNIX Fast File System (UFS/FFS), network sockets API, first widely used TCP/IP stack, FTP, sendmail, cron, vi, BIND, ...
- FreeBSD open-source operating system roughly 20 years old
  - 1993: FreeBSD 1.0 without support for multiprocessing
  - 1998: FreeBSD 3.0 with giant-lock kernel
  - 2003: FreeBSD 5.0 with fine-grained locking
  - 2005: FreeBSD 6.0 with mature fine-grained locking
  - 2012: FreeBSD 9.0 with TCP scalability beyond 32 cores
  - 2013*: FreeBSD 10.0 with non-uniform memory (NUMA)

* Or perhaps early 2014?

Before multiprocessing

- Preemptive multitasking and multithreading for user processes
- Kernel internally multithreaded
  - Represent user threads “in kernel” during system calls/page faults
  - Kernel services utilise threads (e.g., VM, file system, …)
- Most kernel code runs under mutual exclusion
  - Implied condition variables associated with every kernel address
    - `struct foo x;`  
    - `sleep(&x, secs).wakeup(&x)`
  - `lockmgr` reader-writer lock can be held over blocking I/O
  - Sleeping with `lockmgr` or `sleep` triggers context switching
- Critical sections prevent untimely preemption by interrupts
Pre-multiprocessor scheduling

CPU-level synchronisation

- Late 1990s: commodity multi-CPU hardware available from Intel, others
- Architecture-specific atomic operations
  - Compare-and-swap
  - Test-and-set
  - Load linked/store conditional
- Inter-processor interrupts (IPIs)
  - One CPU can trigger an interrupt on another, running handler
- Vendor-specific extensions
  - MIPS inter-thread message passing
  - Intel TM support
Giant locking the kernel

- FreeBSD follows in the footsteps of Cray, Sun, etc.
- Parallel user programs with non-parallel kernel
  - “Giant” spinlock around kernel
  - Acquire on syscall/trap to kernel
  - Drop on return
  - Kernel “migrates” between CPUs on demand
- Interrupts
  - If interrupt delivered on CPU X while kernel is running on CPU Y, forward interrupt to Y

Giant-locked scheduling

Serial kernel execution; parallelism opportunity missed

Kernel-user parallelism

Kernel giant-lock contention
Fine-grained locking

- Giant-locked kernels good for parallel user programs
- But kernel-centred workloads trigger Giant contention
  - E.g., heavy TCP use in web-server workloads
- Motivates move to fine-grained locking
  - FreeBSD adopts *pthreads*-like model for the kernel
- Familiar multi-threading environment
- Mutexes/condition variables rather than semaphores
- Why? Among other things: priority inheritance

Fine-grained scheduling

True kernel-kernel parallelism
Software synchronisation

• Spin locks
• Sleepable locks with different use cases/optimisations
• Mutexes, reader-writer (RW), read-mostly (RM) locks
  • Will sleep for only a **bounded period** of time
• Shared-exclusive (SX) locks
  • May sleep for an **unbounded period** of time
• Implied lock order: unbounded- before bounded-period locks
• Most lock types support priority propagation
• Condition variables, usable with all lock types
  Why? Mutexes are used only for “short” waits, so safe to use them (and wait on them) implementing “long” waits -- e.g., disk I/O

Spinlocks

• Synchronisation internal to the scheduler, interrupts
• E.g., protect sleep queues for mutexes and condition variables
• Spinlock acquire:
  • Disable interrupts
  • Spin on test-and-set to replace `MTX_UNOWNED` with `thread ID`
• Spinlock release:
  • Set lock to `MTX_UNOWNED`
  • Enable interrupts
• More complicated cases involve **lock recursion**
  Interrupt handlers borrow (preempt) contexts synchronously. If a handler tries to acquire a spinlock held by the context it has preempted, deadlock!
Mutexes, RW locks

- Like semaphores, sleep rather than [always] spinning
- Unlike spinlocks, mutexes allow interrupts + preemption
- Implement priority inheritance
- Sleeping is really expensive (scheduler-internal spinlocks)
- Adaptive mutexes address common-case contention
- Spin if the holder of the lock executing on another CPU
- rwlocks are a variation supporting read locking
- Mutexes, rwlocks for most in-kernel synchronisation

Mutex KPIs

- Very similar to pthread mutexes in every way
- struct mtx m;
- void mtx_init(m, name, type, opts)
- void mtx_destroy(m)
- void mtx_lock(m)
- void mtx_unlock(m)
- int mtx_trylock(m)
- void mtx_assert(m)

Notice: no confusing error values from lock and unlock!
Name and type used by WITNESS lock order verifier - more on that later
Condition variables

- Pretty much as we talked about for POSIX - condition variables are used with locks, but not bound to specific monitors
  - `void cv_init(cv, desc)`
  - `void cv_destroy(cv)`
  - `void cv_wait(cv, lo)`
  - `void cv_wait_sig(cv, lo)`
  - `int cv_timedwait(cv, lo)`
  - `int cv_timedwait_sig(cv, lo)`
  - `void cv_signal(cv)`
  - `void cv_broadcast(cv)`

String description allows `ps` to show what CV thread is waiting on - useful for debugging!

Timed waits for I/O timeouts; `_sig` variants interruptible by UNIX signals

`lo` can be any type of lock object, including mutexes, rwlocks, etc.

Scalability

Key idea:
- **speedup**
  
  As we add more parallelism, we would like the system to get faster.

Another key idea:
- **performance collapse**
  
  Sometimes parallelism hurts performance more than it helps due to work distribution overheads, contention
WITNESS

- FreeBSD kernel relies (almost) entirely on lock order to prevent deadlock
- WITNESS is a lock order debugging tool
  - Warns when a deadlock **might** have occurred due to cycles
  - Enabled only in debugging kernels due to expense (~15%+)
- Tracks both statically declared and dynamic lock orders
  - Static orders most commonly *intra-module*
  - Dynamic orders most commonly *inter-module*
- FreeBSD rarely experiences lock-related deadlocks due to partial order
- However, I/O and sleep deadlocks are harder to detect/debug
  - Condition variables make it hard to know what thread is waited on

* Turns out that the global lock order is pretty complicated
Commentary on WITNESS total lock-order graph complexity; courtesy Scott Long, Netflix

* Excerpt from global lock order graph*

This bit of the graph largely relates to networking

Network interface locks: “transmit” tends to occur at the bottom of call stacks via many layers holding locks

Local clusters: e.g., a set of closely related locks from the pf firewall; two are leaf nodes; one is held over calls to another subsystem

UMA zone lock implicitly or explicitly follows most other locks in the system, since almost all components depend on memory allocation

* Turns out that local lock order is pretty complicated too
WITNESS debug output

1st 0xffffffff80025207f0 run0_node_lock (run0_node_lock) @ /usr/src/sys/net80211/ieee80211_ioctl.c:1341
2nd 0xffffffff80025142a8 run0 (network driver) @ /usr/src/sys/modules/usb/run/../../../dev/usb/wlan/if_run.c:3368

KDB: stack backtrace:
db_trace_self_wrapper() at db_trace_self_wrapper+0x2a
kdb_backtrace() at kdb_backtrace+0x37
_witness_debugger() at _witness_debugger+0x2c
witness_checkorder() at witness_checkorder+0x853
_mtx_lock_flags() at _mtx_lock_flags+0x85
run_raw_xmit() at run_raw_xmit+0x85
ieee80211_send_mgmt() at ieee80211_send_mgmt+0x4d5
domlme() at domlme+0x95
setmlme_common() at setmlme_common+0x2f0
ieee80211_ioctl_setmlme() at ieee80211_ioctl_setmlme+0x7e
ieee80211_ioctl_set80211() at ieee80211_ioctl_set80211+0x46f
in_control() at in_control+0xad
if ioctl() at ioctl+0xece
kern_ioctl() at sys_ioctl+0xf0
amd64_syscall() at amd64_syscall+0x380
Xfast_syscall() at Xfast_syscall+0xf7
--- syscall (54, FreeBSD ELF64, sys_ioctl), rip = 0x800de7aec, rsp = 0x7fffffffd848, rbp = 0x2a ---

So how is all this used?

- Kernel is heavily multi-threaded
- Each user thread has a corresponding kernel thread
  - Represents user thread when in syscall, page fault, etc.
- Many kernel services rely on/execute in asynchronous threads
  - Interrupts, timers, I/O, networking, etc.
- Therefore extensive synchronisation
- Locking model is almost always data-oriented
- Think monitors rather than critical sections
- Reference counting or reader-writer locks used for stability
Asynchronous packet processing occurs in a netisr “soft” ithread

Kernel-internel concurrency is represented using a familiar shared memory threading model

Case study: network stack

- First, make it safe without the Giant lock
- Lots of data structures require locks
- Process synchronisation already exists but will be added to
- Establish key work flows, lock orders
- Then, optimise
- Especially locking primitives themselves
- As hardware becomes more parallel, identify and exploit further concurrency opportunities
- Add more threads and distributing more work
Network-stack work flow

- Don’t need to understand details of networking:
  - Applications send and receive data on sockets
  - Packets go in and out of network interface
  - The middle bit of that picture is full of layers
- Processing occurs in *layers*: decapsulation, lookup, reassembly, ...
- Layers are sometimes *directly dispatched* and sometimes involve a *producer-consumer queue* to a second thread
- In latter case, we experience concurrency (even parallelism)
- Send and receive paths also (largely) concurrent

Network stack work flows

Applications send and receive data on sockets

Packets come in … and packets go out.
What to lock and how? (1)

- Fine-grained locking overhead vs. coarse-grained contention
  - Some contention is inevitable: reflects actual communication
  - Other contention is effectively false sharing
- Principle: data locks rather than critical sections
- Key structures: network interfaces, sockets, work queues
- Independent instances should be parallelisable
- Different locks at different layers (sockets vs. control blocks)
- Parallelism at the same layer (receive vs. send socket buffers)
- Things not to lock: mbufs (“work”)

Example: universal memory allocator (UMA)

- Key low-level kernel component
- Slab allocator (Bonwick 1994)
- Object-oriented memory model: init/destroy, alloc/free
- Per-CPU caches
  - Protected by critical sections
  - Encourage locality by allocating memory where last freed
  - Avoid zone lock contention
Work distribution

- Packets are units of work
- Parallel work requires distribution to multiple threads
- Must keep packets ordered -- or TCP gets very upset!
- This requires a strong notion of per-flow serialisation
  - I.e., no generalised producer-consumer/round robin
- Various strategies to keep work ordered – process in a single thread, or multiple threads linked by a queue, etc.
- Establish flow-CPU affinity – utilise caches well

TCP input path

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Kernel</th>
<th>Userspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Linker layer + driver</td>
<td>Socket</td>
</tr>
<tr>
<td>IP</td>
<td>Validate checksum, strip IP header</td>
<td>Reassemble segments, deliver to socket</td>
</tr>
<tr>
<td>TCP + Socket</td>
<td>Look up socket</td>
<td>Kernel copies out mbufs + clusters</td>
</tr>
<tr>
<td>Application</td>
<td>Data stream to application</td>
<td></td>
</tr>
</tbody>
</table>

Potential dispatch points
A more recent trend: multiqueue NICs

- Key source of OS contention: locks around access to hardware devices
- Parallelism for hardware interface: each NIC has $N$ input and output queues
- Flow order maintained by hashing 2- and 4-tuples in TCP/IP headers
- Each input queue assigned its own thread to process

Complex interactions between scheduling and work

<table>
<thead>
<tr>
<th>Processes</th>
<th>1 - multi</th>
<th>2 - single link, proto</th>
<th>3 - single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Net bandwidth in Gb/s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Single-threaded processing caps out a bit over 1Gb/s on this hardware
- Hardware work distribution to multiple threads is a little higher, but more importantly, has lower variance
- Software work distribution to multiple threads gets close to 4Gb/s
- Notice shapes of curves: parallelism helps, but saturation hurts
Changes in hardware motivate changes in concurrency strategy

- Counting instructions → cache misses
- Lock contention → cache line contention
- Locking → find parallelism opportunities
- Work ordering, classification, distribution
- NIC offload of even more protocol layers
- Vertically integrate distribution/affinity

Longer-term strategies

- Optimise for contention: communication is inevitable
- Increase use of lockless primitives: e.g., stats, queues
- Use optimistic techniques for infrequent writes: rmlocks
- Replicate data structures; perhaps with weak consistency
  - E.g., per-CPU statistics, per-CPU memory caches
- Use distribution/affinity strategies minimising contention
- Address not just parallelism, but NUMA and I/O affinity
Conclusion

• FreeBSD employs many of techniques we’ve discussed
  • Mutual exclusion, process synchronisation
  • Producer-consumer
  • Lockless primitives
  • Transaction-like notions – e.g., file system journaling
• But real-world systems are **really** complicated
  • Hopefully you will mostly consume, rather than produce, concurrency primitives like these
• See you in distributed systems!