Concurrent systems
Case study: FreeBSD kernel concurrency

Dr Robert N. M. Watson

FreeBSD kernel

• Open-source OS kernel
  – **Large**: millions of LoC
  – **Complex**: thousands of subsystems, drivers, ...
  – **Very concurrent**: dozens or hundreds of CPU cores/threads
  – **Widely used**: NetApp, EMC, Dell, Apple, Juniper, Netflix, Sony, Cisco, Yahoo!, ...

• Why a case study?
  – Employs C&DS principles
  – Concurrency performance and composability at scale

BSD + FreeBSD: a brief history

• 1980s Berkeley Standard Distribution (BSD)
  – ‘BSD’-style open-source license (MIT, ISC, CMU, ...)
  – UNIX Fast File System (UFS/FFS), sockets API, DNS,
    used TCP/IP stack, FTP, sendmail, BIND, cron, vi, ...
• Open-source FreeBSD operating system
  1993: FreeBSD 1.0 without support for multiprocessing
  1998: FreeBSD 3.0 with “giant-lock” multiprocessing
  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

FreeBSD: before multiprocessing (1)

• Concurrency model inherited from UNIX
• Userspace
  – Preemptive multitasking between processes
  – Later, preemptive multithreading within processes
• Kernel
  – ‘Just’ a C program running ‘bare metal’
  – Internally multithreaded
  – User threads ‘in kernel’ (e.g., in system calls)
  – Kernel services (e.g., async. work for VM, etc.)
FreeBSD: before multiprocessing (2)

- **Cooperative multitasking** within kernel
  - Mutual exclusion as long as you don’t `sleep()`
  - Implied global lock means local locks rarely required
  - Except for interrupt handlers, non-preemptive kernel
  - **Critical sections** control interrupt-handler execution

- **Wait channels**: implied condition variable for every address
  ```c
  sleep(&x, ...); // Wait for event on &x
  wakeup(&x); // Signal an event on &x
  ```
  - Must leave global state consistent when calling `sleep()`
  - Must reload any cached local state after `sleep()` returns

- Use to build higher-level synchronization primitives
  - E.g., `lockmgr()` reader-writer lock can be held over I/O (`sleep`)

---

Pre-multiprocessor scheduling

Lots of unexploited parallelism!
Hardware parallelism, synchronization

- Late 1990s: multi-CPU begins to move down market
  - In 2000s: 2-processor a big deal
  - In 2010s: 64-core is increasingly common
- Coherent, symmetric, shared memory systems
  - Instructions for atomic memory access
    - Compare-and-swap, test-and-set, load linked/store conditional
- Signaling via Inter-Processor Interrupts (IPIs)
  - CPUs can trigger an interrupt handler on each another
- Vendor extensions for performance, programmability
  - MIPS inter-thread message passing
  - Intel TM support: TSX (Whoops: HSW136!)

Giant locking the kernel

- FreeBSD follows footsteps of Cray, Sun, ...
- First, allow user programs to run in parallel
  - One instance of kernel code/data shared by all CPUs
  - Different user processes/threads on different CPUs
- Giant spinlock around kernel
  - Acquire on syscall/trap to kernel; drop on return
  - In effect: kernel runs on at most once CPU at a time;
  - ‘migrates’ between CPUs on demand
- Interrupts
  - If interrupt delivered on CPU X while kernel is on CPU Y, forward interrupt to Y using an IPI
Giant-locked scheduling

Fine-grained locking

- Giant locking is OK for user-program parallelism
- Kernel-centered workloads trigger Giant contention
  - Scheduler, IPC-intensive workloads
  - TCP/buffer cache on high-load web servers
  - Process-model contention with multithreading (VM, ...)
- Motivates migration to fine-grained locking
  - Greater granularity (may) afford greater parallelism
- Mutexes/condition variables rather than semaphores
  - Increasing consensus on pthreads-like synchronization
  - Explicit locks are easier to debug than semaphores
  - Support for priority inheritance + priority propagation
  - E.g., Linux is also now migrating away from semaphores
Fine-grained scheduling

Kernel synchronization primitives

- **Spin locks**
  - Used to implement the scheduler, interrupt handlers
- **Mutexes, reader-writer, read-mostly locks**
  - Most heavily used – different optimization tradeoffs
  - Can be held only over “bounded” computations
  - **Adaptive**: on contention, sleep is expensive; spin first
  - Sleeping depends on scheduler, and hence on spinlocks...
- **Shared-eXclusive (SX) locks, condition variables**
  - Can be held over I/O and other unbounded waits
- **Condition variables** usable with any lock type
- Most primitives support **priority propagation**
WITNESS lock-order checker

- Kernel relies on **partial lock order** to prevent deadlock
  (Recall dining philosophers)
  - In-field lock-related deadlocks are (very) rare
- **WITNESS** is a **lock-order debugging tool**
  - Warns when lock cycles (could) arise by tracking edges
  - Only in debugging kernels due to overhead (15%+)
- Tracks both statically declared, dynamic lock orders
  - **Static orders** most commonly intra-module
  - **Dynamic orders** most commonly inter-module
- Deadlocks for condition variables remain hard to debug
  - What thread should have woken up a CV being waited on?
  - Similar to semaphore problem

WITNESS: global lock-order graph*

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

Excerpt from global lock-order graph*

This bit mostly has to do with networking

Local clusters: e.g., related locks from the firewall: two leaf nodes; one is held over calls to other subsystems

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

Memory allocator locks follow most other locks, since most kernel components require memory allocation

* The local lock-order graph is also complicated.
WITNESS debug output

```
1st 0xffffffff0025207f0 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 
runcmd() at runcmd+0x58 
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 
ifioctl() at ifioctl+0xecf 
kern_ioctl() at kern_ioctl+0xced 
sys_ioctl() at sys_ioctl+0xf0 
amd64_syscall() at amd64_syscall+0x380 
Xfast_syscall() at Xfast_syscall+0xf7 
--- syscall (54, FreeBSD ELF64, sys_ioctl), rip = 0x800de7ae 
0xfffffffff80000000, rbp = 0x2a ---
```

How does this work in practice?

- Kernel is heavily multi-threaded
  - Each user thread has a corresponding kernel thread
    - Represents user thread when in syscall, page fault, etc.
  - Kernels services often execute in asynchronous threads
    - Interrupts, timers, I/O, networking, etc.
- Therefore extensive synchronization
  - Locking model is almost always data-oriented
  - Think ‘monitors’ rather than ‘critical sections’
  - Reference counting or reader-writer locks used for stability
  - Higher-level patterns (producer-consumer, active objects, etc.) used frequently
Kernel threads in action

<table>
<thead>
<tr>
<th>PID</th>
<th>TID</th>
<th>COMM</th>
<th>TDNAME</th>
<th>CPU</th>
<th>PRI</th>
<th>STATE</th>
<th>WCHAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>100003</td>
<td>idle</td>
<td>idle: cpu0</td>
<td>0</td>
<td>255</td>
<td>run</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>100024</td>
<td>intr</td>
<td>irq14: ata0</td>
<td>0</td>
<td>12</td>
<td>wait</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>100025</td>
<td>intr</td>
<td>irq15: ata1</td>
<td>1</td>
<td>12</td>
<td>wait</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>100026</td>
<td>intr</td>
<td>sv1: netisr 0</td>
<td>1</td>
<td>28</td>
<td>wait</td>
<td>-</td>
</tr>
<tr>
<td>3588</td>
<td>10017</td>
<td>sshd</td>
<td>-</td>
<td>0</td>
<td>122</td>
<td>wait</td>
<td>select</td>
</tr>
</tbody>
</table>

- Device-driver interrupts execute in kernel threads
- Asynchronous packet processing occurs in a netisr ‘soft’ thread
- Familiar userspace thread: sshd, blocked in network I/O ('in kernel')
- Idle CPUs are occupied by an idle thread… why?

Case study: the network stack (1)

- What is a network stack?
  - Kernel-resident library of networking routines
  - Sockets, TCP/IP, UDP/IP, Ethernet, ...
- Implements user abstractions, network-interface abstraction, protocol state machines, sockets, etc.
  - System calls: socket(), connect(), send(), recv(), listen(), ...
- Highly complex and concurrent subsystem
  - Composed from many (pluggable) elements
  - Socket layer, network device drivers, protocols, ...
- Typical paths ‘up’ and ‘down’: packets come in, go out
Network-stack work flows

Applications send, receive, await data on sockets

Data/packets processed; dispatched via producer-consumer relationships

Packets go in and out of network interfaces

The work: adding/removing headers, calculating checksums, fragmentation/defragmentation, segment reassembly, reordering, flow control, etc.

Case study: the network stack (2)

• First, make it safe without the Giant lock
  – Lots of data structures require locks
  – Condition signaling already exists but will be added to
  – Establish key work flows, lock orders

• Then, make it fast
  – Especially locking primitives themselves
  – Increase locking granularity where there is contention

• As hardware becomes more parallel, identify and exploit further concurrency opportunities
  – Add more threads, distribute more work
What to lock and how?

- Fine-grained locking **overhead vs. contention**
  - Some contention is **inherent**: reflects necessary communication
  - Some contention is **false sharing**: side effect of structure choices
- Principle: **lock data, not code** (i.e., not critical sections)
  - Key structures: network interfaces, sockets, work queues
  - Independent structure instances often have their own locks
- Horizontal vs. vertical parallelism
  - H: Different locks for different connections (e.g., TCP1 vs. TCP2)
  - H: Different locks within a layer (e.g., receive vs. send buffers)
  - V: Different locks at different layers (e.g., socket vs. TCP state)
- Things not to lock: packets in flight - mbufs ('work')

---

Example: Universal Memory Allocator (UMA)

- Key kernel service
- Slab allocator
  - (Bonwick 1994)
- Per-CPU caches
  - Individually locked
  - Amortise (or avoid) global lock contention
- Some allocation patterns use only per-CPU caches
- Others require dipping into the global pool
Work distribution

- Packets (mbufs) are units of work
- Parallel work requires distribution to threads
  - Must keep packets ordered – or TCP gets cranky!
- Implication: strong per-flow serialization
  - I.e., no generalized producer-consumer/round robin
  - Various strategies to keep work ordered; e.g.:
    - Process in a single thread
    - Multiple threads in a ‘pipeline’ linked by a queue
  - Misordering allowed between flows, just not within them
- Establish flow-CPU affinity can both order processing and utilize caches well

Scalability

Key idea:
- speedup
  As we add more parallelism, we would like the system to get faster.
- performance collapse
  Sometimes parallelism hurts performance more than it helps due to work-distribution overheads, contention.
Longer-term strategies

- Hardware change motivates continuing work
  - Optimize inevitable contention
  - Lockless primitives
  - rmlocks, read-copy-update (RCU)
  - Per-CPU data structures
  - Distribute work to more threads .. to utilise growing core count
- Optimise for locality, not just contention: cache, NUMA, and I/O affinity

Conclusions

- FreeBSD employs many of C&DS techniques
  - Mutual exclusion, condition synchronization
  - Producer-consumer, lockless primitives
  - Also Write-Ahead Logging (WAL) in filesystems
- Real-world systems are really complicated
  - Hopefully, you will mostly consume, rather than produce, concurrency primitives like these
  - Composition is not straightforward
  - Parallelism performance wins are a lot of work
  - Hardware continues to evolve
- See you in Distributed Systems!