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
  - Except for interrupt handlers, non-preemptive kernel
  - Mutual exclusion as long as you don’t sleep()
  - Implied global lock means local locks rarely required
- Wait channels: implied condition variable for every address
  ```
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
- Primitive to build more complex synchronization tools
  - E.g., `lockmgr()` reader-writer lock can be held over I/O (sleep)
- **Critical sections** control interrupt-handler execution

---

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
  - No affinity model: schedule work on first available CPU
- **Giant** spinlock around kernel
  - Acquire on syscall/trap to kernel; drop on return
  - In effect: kernel ‘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

- User-user parallelism
  - Giant locking is fine for user-program parallelism
- Kernel-user 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
- Serial kernel execution; parallelism opportunity missed
  - Parallelism opportunity missed

Fine-grained locking

- Giant locking is fine 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
- Why this approach?
  - Increasing consensus on pthreads-like synchronization
  - Unlike semaphores, access to priority inheritance
Fine-grained scheduling

Kernel synchronization primitives

- **Spin locks** – scheduler, interrupt synchronization
- **Mutexes, reader-writer, read-mostly locks**
  - Most heavily used – different optimization tradeoffs
  - Sleep for only a ‘bounded’ period of time
- **Shared-eXclusive (SX) locks, condition variables**
  - May sleep for an unbounded period of time
  - Implied lock order: unbounded before bounded; why?
- Condition variables usable with any lock type
- **Adaptive**: sleeping is expensive, spin for a bit first
- Most primitives support **priority propagation**
WITNESS lock-order checker

- Kernel relies on **partial lock order** to prevent deadlock (Recall dining philosophers)
- 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
- In-field lock-related deadlocks are (very) rare
- Unbounded sleep (e.g., I/O) deadlocks harder to debug
  - What thread should have woken up a CV being waited on?

WITNESS: global lock-order graph*

* Turns out that the global lock-order graph is pretty complicated.
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” tends to occur at the bottom of call stacks via many layers holding locks

UMA zone lock implicitly or explicitly follows most other locks, since most kernel components depend on memory allocation

* The local lock-order graph is also complicated.
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

Vast hords of threads represent concurrent kernel activities

Idle CPUs are occupied by an idle thread ... why?

Device-driver interrupts execute in kernel **interrupt threads** (ithreads) within kernel-only ‘intr’ process

Asynchronous packet processing occurs in a netiств ‘soft’ ithread

Familiar userspace thread: sshd, blocked in network I/O (‘in kernel!’)

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

---

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, sockets, protocol state machines, 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 ‘down’ and ‘up’: packets come in, go out
Network-stack work flows

Applications send, receive, await data on sockets

Data/packets processed; enqueued at various dispatch or buffering points

Packets go in and out of network interfaces

The work: adding/removing headers, calculating checksums, fragmentation/defragmentation, segment reassembly, ensuring order, flow control, congestion, 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, optimize
  – Especially locking primitives themselves

• 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. coarse-grained contention
  - Some contention is inevitable: reflects need for communication
  - Other 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 instances should be parallelizable
- 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)
- Object-oriented model
  - init/destroy, alloc/free
- Per-CPU caches
  - Protected by critical sections
  - Encourage cache locality by next allocating memory where last freed
  - Avoid zone-lock contention
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
• Establish flow-CPU affinity can both order processing and utilize caches well

TCP input path

- Hardware
  Device: Receive, validate checksum
  Linker layer + driver

- Kernel
  IP: Interpret and strip link layer header
  Validate checksum, strip IP header
  TCP: Validate checksum, strip TCP header
  Look up socket
  Reassemble segments, deliver to socket

- Userspace
  Application: Data stream to application
  Kernel copies out mbufs + clusters
  User thread
  Network software thread
  iThread
### Scalability

What might we expect if we didn’t hit contention?

**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.

### Complex interactions between scheduling and work

#### Varying dispatch strategy — bandwidth

- **Single-threaded processing** caps out a bit over 1Gb/s on this hardware
- **Software work distribution** to multiple threads gets close to 4Gb/s
- **Hardware work distribution** to multiple threads has higher throughput, but more importantly, has lower variance. Why?

Notice shapes of curves: parallelism helps, but saturation hurts
Changes in hardware impact software

• Hardware-design dynamics affect software:
  – Counting instructions $\rightarrow$ cache misses
  – Lock contention $\rightarrow$ cache-line contention
  – Locking $\rightarrow$ find parallelism opportunities
  – Work ordering, classification, distribution
  – NIC offload of even more protocol layers
  – Vertically integrate distribution/affinity
  – DMA/cache interactions

• But: core principles for concurrency control (synchronization) remain the same

Longer-term strategies

• Optimize for inevitable contention
• Lockless primitives
  – E.g., stats, queues
• Tune primitives for workloads
  – E.g., rmlocks, read-copy-update (RCU)
• Replicate data structures; with weak consistency?
  – E.g., per-CPU statistics, per-CPU memory caches
• Distribution/affinity to minimize contention
• From parallelism to NUMA + I/O affinity
Conclusions

• FreeBSD employs many of C&DS techniques
  – Mutual exclusion, process synchronization
  – Producer-consumer
  – Lockless primitives

• 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!