Introduction to Multithreading and Multiprocessing in the FreeBSD SMPng Network Stack

EuroBSDCon 2005
26 November 2005

Robert Watson
FreeBSD Core Team
rwatson@FreeBSD.org

Ed Maste
FreeBSD Developer
emaste@FreeBSD.org

Computer Laboratory
University of Cambridge

Sandvine, Inc.
Introduction

- **Background**
  - Symmetric Multi-Processing (SMP)
  - Strategies for SMP-capable operating systems

- **SMPng Architecture**
  - FreeBSD 3.x/4.x SMP
  - FreeBSD 5.x/6.x SMPng

- **Network Stack**
  - Architecture
  - Synchronization approaches
  - Optimization approaches
Multi-Processing (MP) and Symmetric Multi-Processing (SMP)

- Symmetric Multi-Processing (SMP)
  - More than one general purpose processor
  - Running the same primary system OS
  - Increase available CPU capacity sharing memory/IO resources

- “Symmetric”
  - Refers to memory performance and caching
  - In contrast to NUMA
    - Non-Uniform Memory Access
  - In practice, a bit of both
    - Amd64 NUMA, dual core, etc.
    - Intel HTT, dual core, etc.
Simplified SMP Diagram
Intel Quad Xeon

CPU0
CPU0 Cache

CPU1
CPU1 Cache

CPU2
CPU2 Cache

CPU3
CPU3 Cache

Northbridge

System Memory
Simplified NUMA Diagram
Quad AMD Opteron

CPU0
CPU0 Cache
CPU0 Memory

CPU1
CPU1 Cache
CPU1 Memory

CPU2
CPU2 Cache
CPU2 Memory

CPU3
CPU3 Cache
CPU3 Memory

HT Crossbar / Bus
Not SMPng: Loosely Connected Computation Clusters
What is shared in an SMP System?

<table>
<thead>
<tr>
<th>Shared</th>
<th>Not Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>System memory</td>
<td>CPU (register context, TLB, ...)</td>
</tr>
<tr>
<td>PCI buses</td>
<td>Cache</td>
</tr>
<tr>
<td>I/O channels</td>
<td>Local APIC timer</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

• Sources of asymmetry
  – Hyper-threading (HTT): physical CPU cores share computation resources and caches
  – Non-Uniform Memory Access (NUMA): different CPUs may access regions of memory at different speeds
What is an MP-Capable OS?

- An OS is MP-capable if it is able to operate correctly on MP systems
  - This could mean a lot of different things
  - Usually implies it is able to utilize >1 CPU
- Common approach is Single System Image
  - “Look like a single-processor system”
  - But be faster
- Other models are possible
  - Most carefully select variables to degrade
  - Weak memory models, message passing, ...
OS Approach: Single System Image (SSI)

- To the extent possible, maintain the appearance of a single-processor system
  - Only with more CPU power
- Maintain current UNIX process model
  - Parallelism between processes
  - Parallelism in thread-enabled processes
  - Requires minimal changes to applications yet offer significant performance benefit
- Because the APIs and services weren't designed for MP, not always straightforward
Definition of Success

● Goal is performance
  – Why else buy more CPUs?
  – However, performance is a nebulous concept
    • Very specific to workload
  – Systems programming is rife with trade-offs
● “Speed up”
  – Measurement of workload performance as number of CPUs increase
  – Ratio of score on N processors to score on 1
● Two goals for the OS
  – Don't get in the way of application speed-up
  – Facilitate application speed-up
“Speed-Up”

- “Idealized” performance
- Not realistic
  - OS + application synchronization overhead
  - Limits on workload parallelism
  - Contention on shared resources, such as I/O + bus
Developing an SMP UNIX System

- Two easy steps
  - Make it run
  - Make it run fast

- Well, maybe a little more complicated
  - Start with the kernel
  - Then work on the applications
  - Then repeat until done
Issues relating to MP for UNIX Operating Systems: Kernel

- Bootstrapping
- Inter-processor communication
- Expression of parallelism
- Data structure consistency
- Programming models
- Resource management
- Scheduling work
- Performance
Issues relating to MP for UNIX Operating Systems: Apps

- Application must be able use parallelism
  - OS must provide primitives to support parallel execution
    - Processes, threads
  - OS may do little, some, or lots of the work
    - Network stack
    - File system
  - An MP-capable and MP-optimized thread library is very important
- System libraries and services may need a lot of work to work well with threads
Inter-Processor Communication

- **Inter-Processor Interrupts (IPI)**
  - Wake up processor at boot time
  - Cause a processor to enter an interrupt handler
  - Comes with challenges, such as deadlocks

- **Shared Memory**
  - Kernel memory will generally be mapped identically when the kernel executes on processors
  - Memory is therefore shared, and can be read or written from any processor
  - Requires consistency and synchronization model
  - Atomic operations, higher level primitives, etc.
Expression of Parallelism

- Kernel will run on multiple processors
  - Most kernels have a notion of threads similar to user application threads
  - Multiple execution contexts in a single kernel address space
  - Threads will execute on only one CPU at a time
  - All execution in a thread is serialized with respect to itself
  - Most systems support migration of threads between processors
  - When to migrate is a design choice affecting load balancing and synchronization
Data Consistency

- Some kernel data structures will be accessed from more than one thread at a time
  - Will become corrupted unless access is synchronized
  - “Race Conditions”
- Low level primitives are usually mapped into higher level programming services
  - From atomic operations and IPIs
  - To mutexes, semaphores, signals, locks, ...
  - Lockless queues and other lockless structures
- Choice of model is very important
  - Affects performance and complexity
Data Consistency: Giant Lock Kernels

- Giant Lock Kernels (FreeBSD 3.x, 4.x)
  - Most straightforward approach to MP OS
  - User process and thread parallelism
  - Kernel executes on one processor at a time to maintain kernel programming invariants
    - Only one can enter the kernel at a time
    - Processors spin if waiting for the kernel

- Easy to implement, but lots of “contention”
  - No in-kernel parallelism
Context Switching in a Giant-Locked Kernel

**CPU0**
- `read()`
- Sleep on I/O
- I/O completes
- Giant acquired
- `read()` returns

**CPU1**
- `socket()`
- Giant acquired
- `socket()` returns

- CPUs spinning waiting for Giant to be released by the other CPU

- **Executing in kernel**
- **Running in user space**
- **Waiting on Giant**
- **Idle**
The Problem: Giant Contention

- Contention in a Giant lock kernel occurs when tasks on multiple CPUs compete to enter the kernel
  - User threads performing system calls
  - Interrupt or timer driver kernel activity
- Occurs for workloads using kernel services
  - File system activity
  - Network activity
  - Misc. I/O activity
  - Inter-Process Communication (IPC)
  - Scheduler and context switches
- Also affects UP by limiting preemption
Addressing Contention: Fine-Grained Locking

- Decompose the Giant lock into a series of smaller locks that contend less
  - Typically over “code” or “data”
  - E.g., scheduler lock permits user context switching without waiting on the file system
  - Details vary greatly by OS

- Iterative approach
  - Typically begin with scheduler lock
  - Dependency locking such as memory allocation
  - Some high level subsystem locks
  - Then data-based locking
  - Drive granularity based on observed contention
ContextSwitchingin a Finely Locked Kernel

CPU0
- read()
- Sleep on I/O
- I/O completes
- read() returns
- send()

CPU1
- socket()
- socket() returns
- send()
- Wait on mutex

Socket buffer mutex briefly in contention

Mutex acquired

Executing in kernel
Running in user space
Waiting on mutex
Idle
FreeBSD SMPng Project

- SMPng work began in 2001
  - Present in FreeBSD 5.x, 6.x
- Several architectural goals
  - Adopt more threaded architecture
    - Threads represent possible kernel parallelism
    - Permit interrupts to execute as threads
  - Introduce various synchronization primitives
    - Mutexes, SX locks, rw locks, semaphores, CV's
  - Iteratively lock subsystems and slide Giant off
- Start with common dependencies
  - Synchronization, scheduling, memory allocation, timer events, ...
FreeBSD Kernel

- Several million lines of code
- Many complex subsystems
  - Memory allocation, VM, VFS, network stack, System V IPC, POSIX IPC, ...
- FreeBSD 5.x
  - Most major subsystems except VFS and some drivers execute Giant-free
  - Some network protocols require Giant
- FreeBSD 6.x almost completely Giant-free
  - VFS also executes Giant-free, although some file systems are not
  - Some straggling device drivers require Giant
Network Stack Components

- Over 400,000 lines of code
  - Excludes distributed file systems
  - Excluding device drivers
- Several significant components
  - “mbuf” memory allocator
  - Network device drivers, interface abstraction
  - Protocol-independent routing and event model
  - Link-layer protocols, network-layer protocols
    - Includes IPv4, IPv6, IPSEC, IPX, EtherTalk, ATM
  - Sockets and socket buffers
  - Netgraph extension framework
## Sample Data Flow
### TCP Send and Receive

<table>
<thead>
<tr>
<th>System call and socket</th>
<th>kern_send()</th>
<th>kern_recv()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sosend()</td>
<td>soreceive()</td>
</tr>
<tr>
<td></td>
<td>sbappend()</td>
<td>sbappend()</td>
</tr>
<tr>
<td>TCP</td>
<td>tcp_send()</td>
<td>tcp_reass()</td>
</tr>
<tr>
<td></td>
<td>tcp_output()</td>
<td>tcp_input()</td>
</tr>
<tr>
<td>IP</td>
<td>ip_output()</td>
<td>ip_input()</td>
</tr>
<tr>
<td>Link Layer, Device Driver</td>
<td>ether_output()</td>
<td>ether_input()</td>
</tr>
<tr>
<td></td>
<td>em_start()</td>
<td>em_intr()</td>
</tr>
</tbody>
</table>
Network Stack Threading
UDP Transmit

netblast

- sosend()
- send()
- udp_output()
- ip_output()
- send() returns
- em_start()

em0 ithread

- em_intr() preempts
- em_intr() returns
- em_clean_transmit_intr()
Network Stack Threading
UDP Receive

netreceive
recv()
netreceive blocks
netreceive wakes up
recv() returns
netisr
soreceive()
schednetisr()
swi_net()
udp_input()
recv() returns
em0 ithread
em_intr() preempts
ip_input()
sbappend() sowakeup()
em_intr() returns
idle
em_process_receive_interrupts()
ether_input()
netreceive wakes up
recv() returns
Network Stack Concerns

- **Overhead: Per-packet costs**
  - Network stacks may process millions of PPS
  - Small costs add up quickly if per-packet

- **Ordering**
  - TCP is very sensitive to mis-ordering

- **Optimizations may conflict**
  - Optimizing for latency may damage throughput, and vice versa

- **When using locks, ordering is important**
  - Lock orders prevent deadlock
  - Data passes in various directions through layers
Locking Strategy

• Lock data structures
  – Don't use finer locks than required by UNIX API
    • I.e., parallel send and receive on the same socket is useful, but not parallel send on the same socket
  – Lock references to in-flight packets, not packets
  – Layers have their own locks as objects at different layers have different requirements

• Lock orders
  – Driver locks are leaf locks with respect to stack
  – Protocol drives most inter-layer activity
  – Acquire protocol locks before driver locks
  – Acquire protocol locks before socket locks
  – Avoid lock order issues via deferred dispatch
Network Stack Parallelism

- Network stack was already threaded in 4.x
  - 4.x had user threads, netisr, dispatched crypto
  - 5.x/6.x add ithreads
- Assignment of work to threads
  - Threads involved are typically user threads, netisr, and ithreads
  - Work split over many threads for receive
  - On transmit, work tends to occur in one thread
  - Opportunities for parallelism in receive are greater than in transmit for a single user thread
Approach to Increasing Parallelism

- **Starting point**
  - Assume a Giant-free network stack
  - Select an interesting workload
  - What are remaining source of contention?
  - Where is CPU-intensive activity serialized in a single thread – leading to unbalanced CPU use?

- **Identify natural boundaries in processing**
  - Protocol hand-offs, layer hand-offs, etc
  - Carefully consider ordering considerations

- **Weigh trade-offs, look for amortization**
  - Context switches are expensive
  - Locks are expensive
MP Programming Challenges

- MP programming is rife with challenges
- A few really important ones
  - Deadlock
  - Locking Overhead
  - Event Serialization
Challenge: Deadlock

- “Deadly Embrace”
- Classic deadlocks
  - Lock cycles
  - Any finite resource
- Classic solutions
  - Avoidance
  - Detect + recover
- Avoid live locks!
Deadlock Avoidance in FreeBSD SMPng

- **Hard lock order**
  - Applies to most mutexes and sx locks
  - Disallow lock cycles
  - WITNESS lock verification tool

- **Variable hierarchal lock order**
  - Lock order a property of data structures
  - At any given moment, the lock order is defined
  - However, it may change as data structure changes

- **Master locks**
  - Master lock used to serialize simultaneous access to multiple leaf locks
Lock Order Verification: WITNESS

- Run-time lock order monitor
  - Tracks lock acquisitions
  - Builds graph reflecting order
  - Detects and warns about cycles
- Supports both hard-coded and dynamic discovery of order
- Expensive but useful
Mitigating Locking Overhead

- Amortize cost of locking
  - Avoid multiple lock operations where possible
  - Amortize cost of locking over multiple packets
- Coalesce/reduce number of locks
  - Excessive granularity will increase overhead
  - Combining across layers can avoid lock operations necessitated by to lock order
- Serialization “by thread”
  - Execution of threads is serialized
- Serialization “by CPU”
  - Use of per-CPU data structures and pinning/critical sections
Challenge: Event Serialization

- Ordering of packets is critical to performance
  - TCP will misinterpret reordering as requiring fast retransmit
- Ordering constraints must be maintained across dispatch
- Naïve threading violates ordering
Ensuring Sufficient Ordering

- Carefully select an ordering
  - "Source ordering" is used widely in the stack
    - Polling thread(s)
    - Ithread Direct Dispatch
  - Weakening ordering can improve performance
  - Some forms of parallelism maintain ordering more easily than others
Status of SMPng Network Stack

- FreeBSD 5.x and 6.x largely run the network stack without Giant
  - Some less mainstream components still need it
- From “Make it work” to “Make it work fast!”
  - Many workloads show significant improvements: databases, multi-thread/process TCP use, ...
  - Cost of locking hampers per-packet performance for specific workloads: forwarding/bridging PPS
  - UP performance sometimes sub-optimal
  - Of course, 4.x is the gold standard...
- Active work on performance measurement and optimization currently
Summary

- A lightning fast tour of MP
  - Multi-processor system architectures
  - Operating system interactions with MP
  - SMPng architecture and primitives
- And the network stack on MP
  - The FreeBSD network stack
  - Changes made to the network stack to allow it to run multi-threaded
  - Optimization concerns including locking cost and increasing parallelism
  - Concerns such as packet ordering
Conclusion

- SMPng is present in FreeBSD 5.x, 6.x
  - 5.3-RELEASE the first release with Giant off the network stack by default; 5.4-RELEASE includes stability, performance improvements.
  - 6.x includes substantial optimizations, MPSAFE VFS
- Some URLs:
  - http://www.watson.org/~robert/freebsd/netperf/