Multiprocessor OS
COMP9242 – Advanced Operating Systems

Ihor Kuz | ihor.kuz@data61.csiro.au
S2/2017 Week 11
Announcements

• MyExperience Survey
  • Management obsess over response rates, so please give some feedback on the course.

• Exam
  • Wed 8 Nov to Thurs 9 Nov
  • Suggestion: 10 am to 10 am

• Exam Prep
  • Look at exam prep page early (not ½ hour before lecture)
  • Read through practice paper
  • Will go through example paper next week
  • Be prepared!
Overview

• Multiprocessor OS (Background and Review)
  • How does it work? (Background)
  • Scalability (Review)

• Multiprocessor Hardware
  • Contemporary systems (Intel, AMD, ARM, Oracle/Sun)
  • Experimental and Future systems (Intel, MS, Polaris)

• OS Design for Multiprocessors
  • Guidelines
  • Design approaches
    – Divide and Conquer (Disco, Tesselation)
    – Reduce Sharing (K42, Corey, Linux, FlexSC, scalable commutativity)
    – No Sharing (Barrelfish, fos)
Multiprocessor OS
Uniprocessor OS

CPU

OS

App2

Memory

Run queue
Process control blocks
FS structs

OS data

Application data

App1
App2
App3
App4
Multiprocessor OS

CPU

- App1
- OS

CPU

- App3
- OS

CPU

- App4
- OS

CPU

- App4
- OS

Memory

- OS data
  - Run queue
  - Process control blocks
  - FS structs

- Application data
  - App1
  - App2
  - App3
  - App4
Key design challenges:
- Correctness of (shared) data structures
- Scalability (performance doesn’t suffer)
Correctness of Shared Data

- Concurrency control
  - Locks
  - Semaphores
  - Transactions
  - Lock-free data structures

- We know how to do this:
  - In the application
  - In the OS
Scalability

Speedup as more processors added

Ideal

\[ S(N) = \frac{T_1}{T_N} \]
Scalability

Speedup as more processors added

Reality

\[ S(N) = \frac{T_1}{T_N} \]
Scalability and Serialisation

Parallel Program

Processor 1
- Parallel
- Parallel
- Parallel
- Serial
- Parallel

Processor 2
- Parallel
- Parallel
- Parallel
- Serial
- Parallel

Processor 3
- Parallel
- Parallel
- Parallel
- Serial
Scalability and Serialisation

Remember Amdahl’s law

- Serial (non-parallel) portion: when application not running on all cores
- Serialisation prevents scalability

\[
T_1 = 1 = (1 - P) + P
\]

\[
T_N = (1 - P) + \frac{P}{N}
\]

\[
S(N) = \frac{T_1}{T_N} = \frac{1}{(1 - P) + \frac{P}{N}}
\]

\[
S(\infty) \rightarrow \frac{1}{(1 - P)}
\]
Serialisation

Where does serialisation show up?
• Application (e.g. access shared app data)
• OS (e.g. performing syscall for app) How much time is spent in OS?

Sources of Serialisation

Locking (explicit serialisation)
– Waiting for a lock ➔ stalls self
– Lock implementation:
  – Atomic operations lock bus ➔ stalls everyone waiting for memory
  – Cache coherence traffic loads bus ➔ stalls others waiting for memory

Memory access (implicit)
• Relatively high latency to memory ➔ stalls self

Cache (implicit)
• Processor stalled while cache line is fetched or invalidated
• Affected by latency of interconnect
• Performance depends on data size (cache lines) and contention (number of cores)
More Cache-related Serialisation

False sharing

- Unrelated data structs share the same cache line
- Accessed from different processors

→ Cache coherence traffic and delay

Cache line bouncing

- Shared R/W on many processors
- E.g: bouncing due to locks: each processor spinning on a lock brings it into its own cache

→ Cache coherence traffic and delay

Cache misses

- Potentially direct memory access → stalls self
- When does cache miss occur?
  - Application accesses data for the first time, Application runs on new core
  - Cached memory has been evicted
    - Cache footprint too big, another app ran, OS ran
Multiprocessor Hardware
Multi-What?

- Terminology:
  - core, die (chip), package (module, processor, CPU)
- Multiprocessor, SMP
  - >1 separate processors, connected by off-processor interconnect
- Multithread, SMT
  - >1 hardware threads in a single processing core
- Multicore, CMP
  - >1 processing cores in a single die, connected by on-die interconnect
- Multicore + Multiprocessor
  - >1 multicore dies in a package (multi-chip module), on-processor interconnect
  - >1 multicore processors, off-processor interconnect
- Manycore
  - Lots (>100) of cores
Interesting Properties of Multiprocessors

• Scale and Structure
  • How many cores and processors are there
  • What kinds of cores and processors are there
  • How are they organised (access to IO, etc.)
• Interconnect
  • How are the cores and processors connected
• Memory Locality and Caches
  • Where is the memory
  • What is the cache architecture
• Interprocessor Communication
  • How do cores and processors send messages to each other
Contemporary Multiprocessor Hardware

- **Intel:**
  - Nehalem, Westmere: 10 core, QPI
  - Sandy Bridge, Ivy Bridge: 5 core, ring bus, integrated GPU, L3, IO
  - Haswell (Broadwell): 18+ core, ring bus, transactional memory, slices (EP)
  - Skylake (SP): mesh architecture

- **AMD:**
  - K10 (Opteron: Barcelona, Magny Cours): 12 core, Hypertransport
  - Bulldozer, Piledriver, Steamroller (Opteron, FX)
    - 16 core, Clustered Multithread: module with 2 integer cores
  - Zen: on die NUMA: CPU Complex (CCX) (4 core, private L3)

- **Oracle (Sun) UltraSparc T1,T2,T3,T4,T5 (Niagara), M5,M7**
  - T5: 16 cores, 8 threads/core (2 simultaneous), crossbar, 8 sockets,
  - M8: 32 core, 8 threads, on chip network, 8 sockets, 5GHz

- **ARM Cortex A9, A15 MPCore, big.LITTLE**
  - 4 -8 cores, big.LITTLE: A7 + A15
Scale and Structure

- ARM Cortex A9 MPCore
Scale and Structure

• ARM big.LITTLE
Scale and Structure

• Intel Nehalem
Memory Locality and Caches

- NUMA (Non-Uniform Memory Access)
Interconnect

- AMD Barcelona

SATA  
PCIe  
GbE  

Floppy disk drive
Interconnect (Latency)
Interconnect (Bandwidth)

Node 0 to Node 7 via 2 hops (0 will do 2 hops to access 7)

Bandwidth:
- 3GB/s
- 6GB/s
- 4GB/s-3GB/s
- Unidirectional

From https://www.usenix.org/conference/atc15/technical-session/presentation/lepers
Interconnect

- Oracle Sparc T2

From Sun/Oracle
Interconnect

Haswell EP Die Configurations

Not representative of actual die-sizes, orientation and layouts – for informational use only.

<table>
<thead>
<tr>
<th>Chop</th>
<th>Columns</th>
<th>Home Agents</th>
<th>Cores</th>
<th>Power (W)</th>
<th>Transistors (B)</th>
<th>Die Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCC</td>
<td>4</td>
<td>2</td>
<td>14-18</td>
<td>110-145</td>
<td>5.69</td>
<td>662</td>
</tr>
<tr>
<td>MCC</td>
<td>3</td>
<td>2</td>
<td>6-12</td>
<td>65-160</td>
<td>3.84</td>
<td>492</td>
</tr>
<tr>
<td>LCC</td>
<td>2</td>
<td>1</td>
<td>4-8</td>
<td>55-140</td>
<td>2.60</td>
<td>354</td>
</tr>
</tbody>
</table>

From http://www.anandtech.com/show/8423/intel-xeon-e5-version-3-up-to-18-haswell-ep-cores-/4
Cluster on Die (COD) Mode

- Supported on 1S & 2S SKUs with 2 Home Agents (10+ cores)
- In memory directory bits & directory cache used on 2S to reduce coherence traffic and cache-to-cache transfer latencies
- Targeted at NUMA optimized workloads where latency is more important than sharing across Caching Agents
  - Reduces average LLC hit and local memory latencies
  - HA sees most requests from reduced set of threads potentially offering higher effective memory bandwidth
- OS/VMM own NUMA and process affinity decisions
Experimental/Future Multiprocessor Hardware

- Microsoft Beehive
  - Ring bus, no cache coherence
- Tilera (now Mellanox) Tile64, Tile-Gx
  - 100 cores, mesh network
- Intel Polaris
  - 80 cores, mesh network
- Intel SCC
  - 48 cores, mesh network, no cache coherency
- Intel MIC (Multi Integrated Core)
  - Knight’s Corner/Landing - Xeon Phi
  - 60+ cores, ring bus/mesh
Scale and Structure

- Tilera Tile64 (newest: Mellanox TILE-Gx), Intel Polaris
Cache and Memory and IPC

• Intel SCC
Interprocessor Communication

• Beehive
Interconnect

- Intel MIC (Multi Integrated Core) (Knight’s Corner/Landing - Xeon Phi)

From http://semiaccurate.com/2012/08/28/intel-details-knights-corner-architecture-at-long-last/
Summary

• Scalability
  • 100+ cores
  • Amdahl’s law really kicks in

• Heterogeneity
  • Heterogeneous cores, memory, etc.
  • Properties of similar systems may vary wildly (e.g. interconnect topology and latencies between different AMD platforms)

• NUMA
  • Also variable latencies due to topology and cache coherence

• Cache coherence may not be possible
  • Can’t use it for locking
  • Shared data structures require explicit work

• Computer is a distributed system
  • Message passing
  • Consistency and Synchronisation
  • Fault tolerance
OS DESIGN for Multiprocessors
Optimisation for Scalability

• Reduce amount of code in critical sections
  • Increases concurrency
  • Fine grained locking
    – Lock data not code
    – Tradeoff: more concurrency but more locking (and locking causes serialisation)
  • Lock free data structures

• Avoid expensive memory access
  • Avoid uncached memory
  • Access cheap (close) memory
Optimisation for Scalability

• Reduce false sharing
  • Pad data structures to cache lines

• Reduce cache line bouncing
  • Reduce sharing
  • E.g: MCS locks use local data

• Reduce cache misses
  • Affinity scheduling: run process on the core where it last ran.
  • Avoid cache pollution
OS Design Guidelines for Modern (and future) Multiprocessors

• Avoid shared data
  • Performance issues arise less from lock contention than from data locality
• Explicit communication
  • Regain control over communication costs (and predictability)
  • Sometimes it’s the only option
• Tradeoff: parallelism vs synchronisation
  • Synchronisation introduces serialisation
  • Make concurrent threads independent: reduce crit sections & cache misses
• Allocate for locality
  • E.g. provide memory local to a core
• Schedule for locality
  • With cached data
  • With local memory
• Tradeoff: uniprocessor performance vs scalability
Design approaches

• Divide and conquer
  • Divide multiprocessor into smaller bits, use them as normal
  • Using virtualisation
  • Using exokernel

• Reduced sharing
  • Brute force & Heroic Effort
    – Find problems in existing OS and fix them
    – E.g Linux rearchitecting: BKL -> fine grained locking
  • By design
    – Avoid shared data as much as possible

• No sharing
  • Computer is a distributed system
    – Do extra work to share!
Divide and Conquer

Disco

• Scalability is too hard!
• Context:
  • ca. 1995, large ccNUMA multiprocessors appearing
  • Scaling OSes requires extensive modifications
• Idea:
  • Implement a scalable VMM
  • Run multiple OS instances
• VMM has most of the features of a scalable OS:
  • NUMA aware allocator
  • Page replication, remapping, etc.
• VMM substantially simpler/cheaper to implement
• Modern incarnations of this
  • Virtual servers (Amazon, etc.)
  • Research (Cerberus)
Disco Architecture

[Bugnion et al., 1997]
Disco Performance

Normalized Execution Time

<table>
<thead>
<tr>
<th></th>
<th>pmake</th>
<th>RADIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRIX</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1VM</td>
<td>136</td>
<td>34</td>
</tr>
<tr>
<td>2VM</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>4VM</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>8VM</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>8VM/nfs</td>
<td>86</td>
<td></td>
</tr>
</tbody>
</table>

Legend:
- Idle
- Disco
- Sync
- Kernel
- User_stall
- User
Space-Time Partitioning

Tessellation

- Space-Time partitioning
- 2-level scheduling

- Context:
  - 2009-... highly parallel multicore systems
  - Berkeley Par Lab

Tessellation: Space-Time Partitioning in a Manycore Client OS [Liu et al., 2010]
http://tessellation.cs.berkeley.edu/
Tessellation
Reduce Sharing

K42

• Context:
  • 1997-2006: OS for ccNUMA systems
  • IBM, U Toronto (Tornado, Hurricane)

• Goals:
  • High locality
  • Scalability

• Object Oriented
  • Fine grained objects

• Clustered (Distributed) Objects
  • Data locality

• Deferred deletion (RCU)
  • Avoid locking

• NUMA aware memory allocator
  • Memory locality

Clustered Objects, Ph.D. thesis [Appavoo, 2005]
http://www.research.ibm.com/K42/
K42: Fine-grained objects

Traditional System  |  OO Decomposed System

User-level requests

System paths & data structures used to satisfy requests

- much sharing

- much less sharing
- better performance

[Appavoo, 2005]
K42: Clustered objects

- Globally valid object reference
- Resolves to
  - Processor local representative
- Sharing, locking strategy local to each object
- Transparency
  - Eases complexity
  - Controlled introduction of locality
- Shared counter:
  - \textit{inc, dec}: local access
  - \textit{val}: communication
- Fast path:
  - Access mostly local structures
K42 Performance

![Graph showing performance comparison across different types of objects (Linux 2.4.19, K42 Shared VM Objects, K42 Distributed VM Objects) across different numbers of processors. The graph plots throughput against the number of processors.](image-url)
Corey

- **Context**
  - 2008, high-end multicore servers, MIT
- **Goals:**
  - Application control of OS sharing
- **OS**
  - Exokernel-like, higher-level services as libraries
  - By default only single core access to OS data structures
  - Calls to control how data structures are shared
- **Address Ranges**
  - Control private per core and shared address spaces
- **Kernel Cores**
  - Dedicate cores to run specific kernel functions
- **Shares**
  - Lookup tables for kernel objects allow control over which object identifiers are visible to other cores.

Corey: An Operating System for Many Cores [Boyd-Wickizer et al., 2008]
http://pdos.csail.mit.edu/corey
Linux Brute Force Scalability

• Context
  • 2010, high-end multicore servers, MIT

• Goals:
  • Scaling commodity OS

• Linux scalability
  • (2010 – scale Linux to 48 cores)

**Y-axis:** (throughput with 48 cores) / (throughput with one core)
Linux Brute Force Scalability

- Apply lessons from parallel computing and past research
  - sloppy counters,
  - per-core data structs,
  - fine-grained lock, lock free,
  - cache lines
  - 3002 lines of code changed

<table>
<thead>
<tr>
<th>Mount tables</th>
<th>memcached</th>
<th>Apache</th>
<th>Exim</th>
<th>PostgreSQL</th>
<th>gmake</th>
<th>Psearchy</th>
<th>Metis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open file table</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sloppy counters</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inode allocation</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock-free dentry lookup</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Super pages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMA buffer allocation</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network stack false sharing</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel accept</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application modifications</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

- Conclusion:
  - no scalability reason to give up on traditional operating system organizations just yet.
Scalability of the API

• Context
  • 2013, previous multicore projects at MIT

• Goals
  • How to know if a system is really scalable?

• Workload-based evaluation
  • Run workload, plot scalability, fix problems
  • Did we miss any non-scalable workload?
  • Did we find all bottlenecks?

• Is there something fundamental that makes an system non-scalable?
  • The interface might be a fundamental bottleneck
Scalable Commutativity Rule

• The Rule
  • *Whenever interface operations commute, they can be implemented in a way that scales.*

• Commutative operations:
  • Cannot distinguish order of operations from results
  • Example:
    – Creat:
      – Requires that lowest available FD be returned
      – Not commutative: can tell which one was run first

• Why are commutative operations scalable?
  • results independent of order ⇒ communication is unnecessary
  • without communication, no conflicts

• Informs software design process
  • Design: design guideline for scalable interfaces
  • Implementation: clear target
  • Test: workload-independent testing
Commuter: An Automated Scalability Testing Tool

Symbolic model

Analyzer

Commutativity conditions

Testgen

Test cases

Linux

Mtrace/QEMU

Conflicting cache lines

(Linux 3.8, ramfs)

All tests conflict-free

All tests conflicted

Zero cache lines, shared

All tests conflict-free

All tests conflicted

(sv6)
FlexSC

• Context:
  • 2010, commodity multicores
  • U Toronto

• Goal:
  • Reduce context switch overhead of system calls

• Syscall context switch:
  • Usual mode switch overhead
  • But: cache and TLB pollution!

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Instructions</th>
<th>Cycles</th>
<th>IPC</th>
<th>i-cache</th>
<th>d-cache</th>
<th>L2</th>
<th>L3</th>
<th>d-TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>stat</td>
<td>4972</td>
<td>13585</td>
<td>0.37</td>
<td>32</td>
<td>186</td>
<td>660</td>
<td>2559</td>
<td>21</td>
</tr>
<tr>
<td>pread</td>
<td>3739</td>
<td>12300</td>
<td>0.30</td>
<td>32</td>
<td>294</td>
<td>679</td>
<td>2160</td>
<td>20</td>
</tr>
<tr>
<td>pwrite</td>
<td>5689</td>
<td>31285</td>
<td>0.18</td>
<td>50</td>
<td>373</td>
<td>985</td>
<td>3160</td>
<td>44</td>
</tr>
<tr>
<td>open+close</td>
<td>6631</td>
<td>19162</td>
<td>0.34</td>
<td>47</td>
<td>240</td>
<td>900</td>
<td>3534</td>
<td>28</td>
</tr>
<tr>
<td>mmap+munmap</td>
<td>8977</td>
<td>19079</td>
<td>0.47</td>
<td>41</td>
<td>233</td>
<td>869</td>
<td>3913</td>
<td>7</td>
</tr>
<tr>
<td>open+write+close</td>
<td>9921</td>
<td>32815</td>
<td>0.30</td>
<td>78</td>
<td>481</td>
<td>1462</td>
<td>5105</td>
<td>49</td>
</tr>
</tbody>
</table>
FlexSC

• Asynchronous system calls
  • Batch system calls
  • Run them on dedicated cores

• FlexSC-Threads
  • M on N
  • M >> N
FlexSC Results

Apache
FlexSC: batching, sys call core redirect
No sharing

- Multikernel
  - Barrelfish
  - fos: factored operating system
Barrelfish

- **Context:**
  - 2007 large multicore machines appearing
  - 100s of cores on the horizon
  - NUMA (cc and non-cc)
  - ETH Zurich and Microsoft

- **Goals:**
  - Scale to many cores
  - Support and manage heterogeneous hardware

- **Approach:**
  - Structure OS as *distributed system*

- **Design principles:**
  - Interprocessor communication is explicit
  - OS structure hardware neutral
  - State is replicated

- **Microkernel**
  - Similar to seL4: capabilities
Barrelfish

User space:
- Monitor
- CPU driver
- x86-64 CPU / APIC MMU

Kernel space:
- Monitor
- CPU driver
- x86-64 CPU / APIC MMU

Hardware:
- Monitor
- CPU driver
- x86-64 CPU / APIC MMU

URPC
Send IPI
Cache-coherence, Interrupts
Barrelfish: Replication

• Kernel + Monitor:
  • Only memory shared for message channels
• Monitor:
  • Collectively coordinate system-wide state
• System-wide state:
  • Memory allocation tables
  • Address space mappings
  • Capability lists
• What state is replicated in Barrelfish
  • Capability lists
• Consistency and Coordination
  • Retype: two-phase commit to globally execute operation in order
  • Page (re/un)mapping: one-phase commit to synchronise TLBs
Barrelfish: Communication

- Different mechanisms:
  - Intra-core
    - Kernel endpoints
  - Inter-core
    - URPC

- URPC
  - Uses cache coherence + polling
  - Shared buffer
    - Sender writes a cache line
    - Receiver polls on cache line
    - (last word so no part message)
  - Polling?
    - Cache only changes when sender writes, so poll is cheap
    - Switch to block and IPI if wait is too long.
Barrelfish: Results

- Message passing vs caching

![Graph showing latency (cycles x 1000) vs cores for different message passing and caching conditions.](image-url)
Barrelfish: Results

- Broadcast vs Multicast

![Graph showing latency vs cores for broadcast, unicast, multicast, and NUMA-Aware multicast.](image)
Barrelfish: Results

- TLB shootdown
Summary
Summary

• Trends in multicore
  • Scale (100+ cores)
  • NUMA
  • No cache coherence
  • Distributed system
  • Heterogeneity

• OS design guidelines
  • Avoid shared data
  • Explicit communication
  • Locality

• Approaches to multicore OS
  • Partition the machine (Disco, Tessellation)
  • Reduce sharing (K42, Corey, Linux, FlexSC, scalable commutativity)
  • No sharing (Barrelfish, fos)