Multiple processor systems
8.1, 8.2 – Tanenbaum

Marie Roch
contains slides from:
Tanenbaum 3rd ed. © 2008

Multiple CPU systems

• Why multiple CPUS?
  – Some programs are inherently (or embarrassingly) parallel
  – Approaching size/temperature tradeoff limit for current technologies
Multiple CPU systems

**Multiprocessor**
- Each processor can address every word of memory
- Two paradigms
  - Unified memory architecture (UMA)
  - Nonunified memory architecture (NUMA)

**Multicomputer**
- Tightly coupled
- 20-50x10^3 ns access

**Distributed**
- Loosely coupled, 10-100x10^6 ns access

Fig. 8-1 Types of multiple processor systems and typical times required to access shared/remote memory [Tanenbaum, p. 525]
Multiprocessor
UMA cache coherency

• Multicore
  – single cache (e.g. Intel dual core)
  – or multiple cache (e.g. AMD Opteron)

• Multiple caches need a cache coherence protocol (overview)
  – Mark blocks
    • read-only – may be in multiple caches
    • read/write – only in one cache
  – When accessing a word in another CPU’s read/write cache, other cache must write before access completes.

UMA architectures

• single bus
  > 16-32 CPUs ➔ Ouch!

• crossbar switch

• multistage switch
Single bus UMA

- Primary problem: bus contention
- Ways to remedy
  - local cache
  - local private memory

Crossbar switch

Figure 8-3. (a) An 8 × 8 crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.
Crossbar and multistage

• Crossbars
  – grow exponentially
  – are nonblocking (>1 CPU can access the same path to memory)

• Multistage switches provide alternative to exponential growth

• Accessing multistage switches
  – consider read/write as message on memory bus

Omega Network

(an inexpensive multistage switch)

• Each stage is a series of 2x2 switches

• Interconnected in a perfect shuffle…
Figure 8-5. An omega switching network.

Question: How do we get the result?

Omega network

- Blocking network
  (other multistages may differ)
- Interleaving memory
  - consecutive words in different RAM modules
  - prevents tying up any single path
  - can permit parallel access in some architectures
NUMA – non-uniform mem. access

• Interconnect networks become unwieldy
  > ~100 CPUs
• Dual class memory
  – fast local: as usual
  – slow remote: load/store operations only
• Otherwise transparent to user

NUMA flavors

• No Cache (NC-NUMA)
  – Remote memory is not cached
• Cache-coherent (CC-NUMA)
  – Allows cached remote memory
  – Frequently uses directory database for each cache line
    • status (clean/dirty)
    • which cache
CC-NUMA example

- 32 bit address space
- Cache
  - line size: 64 \( (2^6) \) bytes
  - \( 2^{32}/2^6 = 2^{26} \) lines
- 256 single CPU nodes
  - \( 2^{24} \) bytes (16 Mb) local RAM
  - \( 2^{24}/2^6 = 2^{18} \) cache entries
- Addressing scheme

![Addressing scheme diagram](Tanenbaum p. 532)

Suppose node 0 fetches 0xFF0AB004
CC-NUMA example

Node 0 sends message to node FF requesting block 0x02AC0

CC-NUMA example

Case 1: invalid

Case 2: valid
CC-NUMA example

Node FF directory

Case 1: invalid
0x02AC0  0  0x3c

1. Fetch cache line 0x02AC0
2. Send cache line to node 00
3. Update directory to indicate cache line 0x02AC0 at node 0

Case 2: valid
0x02AC0  1  0xEE

1. Send message to node 0xEE
2. Node 0xEE sends line to 0x00
3. Node 0xEE invalidates cache line
4. Node 00 receives cache line
5. Update node 0xFF directory to reflect line now at node 0x00
**CC-NUMA**

- Acceptable overhead
  - \(2^{18}\) high speed (\$\$\$) 9 bit directory entries
  - \(~1.76\%\) for 16 MB RAM

- More sophisticated (expensive) designs let one have multiple caches.

---

**Multicore chips**

- Common RAM for all cores (UMA)
- Common or separate cache

![Diagram of multicore chips]
Multicore chips

• Snooping logic
  – Watch thy neighbor’s writes
  – On write, invalidates all shared instances to ensure cache consistency

• What type of core?
  – homogeneous: same processor
  – heterogeneous: typically system on a chip

Multiprocessor OS

• Separate OS
• Master-slave
• Symmetric multiprocessor

[Diagram: Multiprocessor system with shared memory accessed in 2-10 ns]
Separate OS

- CPUs function as separate computers
- Resources partitioned
  (some sharing possible, e.g. OS code)
- Many details to consider, e.g. …
  - dirty disk pages
  - no easy way to load balance

Master-Slave

- Asymmetric
- OS runs on a specific CPU
Symmetric multiprocessor (SMP)

- OS can be executed by any CPU

- Concurrency issues
  
  Note: race conditions can occur on asymmetric OS as well...

Symmetric multiprocessor (SMP)

- One critical region vs. multiple...

- Deadlocks...

- Remember these issues are also concerns for a multi-threaded kernel on an asymmetric multiprocessor
Multiprocessor synchronization

• Mutual exclusion protocol
  – needs atomic instruction, e.g. TSL/SWAP
  – any atomic instruction must be able to lock the bus
  – what happens if the bus is not locked?

Multiprocessor synchronization

• Playing ping-pong with the cache

Cache 0: lock holder
1. 0x8A3 cached
2. TSL writes, 0x8A3 moves here
3. modifies shared vars, cache moves here
4. Still polling, cache moves here

Cache 1: TSL poller

RAM
Entry:
TSL r0, LOCK
CMP r0, 0
BNE Entry
Mutual Exclusion Code
...
LOCK variable
shared variables

Corresponds to cache line 0x8A3
Multiprocessor synchronization

Strategies to prevent cache invalidation
1. Poll w/ read, use TSL once free
2. Exponential backoff (developed for Ethernet)
3. Grant private lock

When to block

- Spin locks waste CPU cycles, but so do context switches
  - sample context switch: 1 ms
  - sample mutual exclusion: 50 µs
- Mutual exclusion time is unknown…
- Alternatives
  - always spin
  - always switch
  - predict based on history or static threshold
- Does it make sense to spin on a uniprocessor?
Multiprocessor scheduling

- Kernel-level threads
  - Which thread to run?
  - What might influence the decision?

- Which CPU to schedule?

- Timesharing vs. spacesharing

Independent vs. dependent threads

- Independent – unrelated

- Dependent
  - Could be related through a graph
  - May not make as much sense to schedule independently

[Image: Diagram of a multiprocessor system with shared memory.]

[Tanenbaum, p. 525]
Common queue for independent threads

Alternatives/Enhancements

• Smart scheduling
  – thread sets critical section flag
  – extend time quantum when flag set

• Affinity scheduling
  – At CPU burst completion, thread has many cache entries
  – Scheduling soon on same CPU may result in more hits
  – Can assign to CPU, then schedule (2 level scheduling)
Space sharing

- Some processes may benefit from being scheduled simultaneously.
- Typically scheduled FCFS

Gang scheduling

- Space sharing eliminated context switches
- Discretize scheduling
- Spaceshare “gang” of related processes at each interval.
- CPUs remain idle until next quantum if CPU burst completes.
Multicomputers
aka: cluster computers, cluster of workstations

• Recall:
  – Tightly coupled
  – No shared memory

• Nodes
  – CPU (possibly multicore)
  – high speed network
  – RAM
  – perhaps secondary storage

Interconnect

• Various network topologies
• Samples:
  – double torus
  – cube
  – 4D hypercube

• Others: star, ring, grid
Routing

• Packet switched
  – messages packetized
  – “store and forward:” each switch point
    • receives packet
    • forwards to next switch point
    • latency increases with # switch points

• Circuit switched
  – Establish path
  – All bits sent along path

Network interfaces

• Copying buffers increases delay

• Map hardware buffer into user space?
  – problems for multiple users
  – problems for kernel processes

  – partial solution: Use two network interfaces
User-level communication

- Message passing (CS570)
  - send/receive
  - ports/mailboxes
  - addressing
    - unlike Internet, fixed network
    - typically CPU# & port/mailbox or process#
  - blocking/nonblocking

Implementation non-blocking messages

- send
  - user cannot modify buffer until message actually sent
  - three possibilities
    - block until kernel can copy to an internal buffer*
    - generate interrupt once buffer is sent
    - mark page as copy-on-write until sent

* From a network perspective, this call is still non-blocking, but not from an OS one. It is the easiest and most common option implemented.
Receive

- Will blocking cause problems?

- Non-blocking
  - polling
  - message arrival by interrupt
    - inform calling thread (traditional)
    - pop-up threads
    - active messages (pop-up variant)
      - call from user-level interrupt handler

Remote procedure calls (RPCs)

Marshalling - Packing arguments into a message.

Stub functions hide message transmission
RPC Gotchas

• Pointers to data structures that are not well contained (e.g. graph)
• Weak types (e.g. int x[] in C/C++)
• Types can be difficult to deduce (e.g. printf)
• References to globals

Distributed shared memory (DSM)

• Transparent to user
• Modifications to page table
  – Invalid pages may be on another processor
  – Page fault results in fetching page from other CPU’s memory
• Read-only pages can be shared
• Extensions possible (e.g. share until write)
DSM Issues

- Network startup is expensive
- Small difference between sending 1 page vs. 4 pages
- Too large a transfer is more likely to result in false sharing

Sequential consistency

- Suppose we let writable pages be shared:

What if processes 0 and 1 both write to the same location...
Easy sequential consistency

- Mark shared pages read only
- Writing causes a page fault
- Page fault handler
  - send message to other processors to invalidate shared page
  - mark page read/write
  - instruction restart

A little bit trickier…

- Shared pages
  - read/write
  - writes
    - obtain mutex from OS covering region of page
    - write
    - upon release, OS propagates region to other processors
- Other techniques possible…
Multicomputer scheduling

- Admission scheduler is important but easily managed
- Short-time scheduler
  - Any appropriate scheduler for local processes
  - Even multiprocessor algorithms can be considered within node
  - Globally
    - more difficult
    - one possibility: gang scheduling
- Load balancing
  - Plays role of memory scheduler
  - Referred to as a processor allocation algorithm
  - Migrating is expensive

Processor allocation algorithms

- Graph theoretic
- Distributed heuristics
  - distribute work from overloaded nodes
  - solicit work on underloaded nodes
Using graph theory to load balance

Partition graph to minimize internode traffic
How: Beyond the scope of lecture, but it might make a good presentation…

Sender-Initiated Distributed Heuristic Algorithm

When above threshold:
- Solicit peer at random to take processes
- Peer accepts/rejects based upon acceptance threshold
- Up to N probes before running anyway
Receiver-Initiated Distributed Heuristic Algorithm

Question: Both algorithms can result in lots of messages being sent. Which one might perform better?