Learning Outcomes
• An understanding of the structure and limits of multiprocessor hardware.
• An appreciation of approaches to operating system support for multiprocessor machines.
• An understanding of issues surrounding and approaches to construction of multiprocessor synchronisation primitives.

Multiprocessor System
• We will look at shared-memory multiprocessors
• More than one processor sharing the same memory
• A single CPU can only go so fast
• Use more than one CPU to improve performance
• Assumptions
  • Workload can be parallelised
  • Workload is not I/O-bound or memory-bound
• Disks and other hardware can be expensive
  • Can share hardware between CPUs

Amdahl’s law
• Given a proportion $P$ of a program that can be made parallel, and the remaining serial portion $(1-P)$, speedup by using $N$ processors

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

1 Processor Serial Time 50 Parallel
2 Processors Serial 50 Parallel Time

Amdahl’s law
• Given a proportion $P$ of a program that can be made parallel, and the remaining serial portion $(1-P)$, speedup by using $N$ processors

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

1 Processor Serial Time 50 Parallel
2 Processors Serial 50 Parallel Time
Types of Multiprocessors (MPs)

- **UMA MP**
  - Uniform Memory Access
  - Access to all memory occurs at the same speed for all processors.
- **NUMA MP**
  - Non-uniform memory access
  - Access to some parts of memory is faster for some processors than other parts of memory.
- We will focus on UMA

**Bus Based UMA**

- Simplest MP is more than one processor on a single bus connect to memory
  - Bus bandwidth becomes a bottleneck with more than just a few CPUs

**Cache Consistency**

- Cache consistency is usually handled by the hardware.
  - Writes to one cache propagate to, or invalidate appropriate entries on other caches.
  - Cache transactions also consume bus bandwidth

**Multi-core Processor**
Bus Based UMA

- With only a single shared bus, scalability can be limited by the bus bandwidth of the single bus
  - Caching only helps so much
  - Alternative bus architectures do exist.
    - They improve bandwidth available
    - Don’t eliminate constraint that bandwidth is limited

Summary

- Multiprocessors can
  - Increase computation power beyond that available from a single CPU
  - Share resources such as disk and memory

- However
  - Assumes parallelizable workload to be effective
  - Assumes not I/O bound
  - Shared buses (bus bandwidth) limits scalability
    - Can be reduced via hardware design
    - Can be reduced by carefully crafted software behaviour
      - Good cache locality together with limited data sharing where possible

Question

- How do we construct an OS for a multiprocessor?
  - What are some of the issues?

Each CPU has its own OS?

- Statically allocate physical memory to each CPU
- Each CPU runs its own independent OS
- Share peripherals
- Each CPU (OS) handles its processes system calls

Each CPU has its own OS

- Used in early multiprocessor systems to ‘get them going’
  - Simpler to implement
  - Avoids CPU-based concurrency issues by not sharing
  - Scales – no shared serial sections
  - Modern analogy, virtualisation in the cloud.

Issues

- Each processor has its own scheduling queue
  - We can have one processor overloaded, and the rest idle
- Each processor has its own memory partition
  - We can a one processor thrashing, and the others with free memory
    - No way to move free memory from one OS to another
Symmetric Multiprocessors (SMP)

• OS kernel run on all processors
  • Load and resource are balance between all processors
  • Including kernel execution
• Issue: Real concurrency in the kernel
  • Need carefully applied synchronisation primitives to avoid disaster

• One alternative: A single mutex that make the entire kernel a large critical section
  • Only one CPU can be in the kernel at a time
  • The “big lock” becomes a bottleneck when in-kernel processing exceeds what can be done on a single CPU

• Better alternative: identify largely independent parts of the kernel and make each of them their own critical section
  • Allows more parallelism in the kernel
• Issue: Difficult task
  • Code is mostly similar to uniprocessor code
  • Hard part is identifying independent parts that don’t interfere with each other
    • Remember all the inter-dependencies between OS subsystems.

Example:
• Associate a mutex with independent parts of the kernel
  • Some kernel activities require more than one part of the kernel
    • Need to acquire more than one mutex
    • Great opportunity to deadlock!!!!!
  • Results in potentially complex lock ordering schemes that must be adhered to

Given a “big lock” kernel, we divide the kernel into two independent parts with a lock each
• Good chance that one of those locks will become the next bottleneck
• Leads to more subdivision, more locks, more complex lock acquisition rules
• Subdivision in practice is (in reality) making more code multithreaded (parallelised)

Real life Scalability Example
• Early 1990’s, CSE wanted to run 80 X-Terminals off one or more server machines
• Winning tender was a 4-CPU bar-fridge-sized machine with 256M of RAM
  • Eventual config 6-CPU and 512M of RAM
  • Machine ran fine in all pre-session testing
Real life Scalability Example

• Students + assignment deadline = machine unusable

Real life Scalability Example

• To fix the problem, the tenderer supplied more CPUs to improve performance (number increased to 8)
  • No change????
  • Eventually, machine was replaced with
    • Three 2-CPU pizza-box-sized machines, each with 256M RAM
    • Cheaper overall
    • Performance was dramatically improved!!!!!
    • Why?

Lesson Learned

• Building scalable multiprocessor kernels is hard
• Lock contention can limit overall system performance

SMP Linux similar evolution

• Linux 2.0 Single kernel big lock (1996)
• Linux 2.2 Big lock with interrupt handling locks
• Linux 2.4 Big lock plus some subsystem locks
• Linux 2.6 most code now outside the big lock, data-based locking, lots of scalability tuning, etc, etc...
• Big lock removed in 2011 in kernel version 2.6.39

Multiprocessor Synchronisation

• Given we need synchronisation, how can we achieve it on a multiprocessor machine?
  • Unlike a uniprocessor, disabling interrupts does not work.
  • It does not prevent other CPUs from running in parallel
  • Need special hardware support
Recall Mutual Exclusion with Test-and-Set

- **enter_region:**
  ```
  TSL REGISTER,LOCK | copy lock to register and set lock to 1
  CMP REGISTER,#0 | was lock zero?
  JNE enter_region | if it was non zero, lock was set, so loop
  RET | return to caller, critical region entered
  ```

- **leave_region:**
  ```
  MOVE LOCK,#0 | store a 0 in lock
  RET | return to caller
  ```

Entering and leaving a critical region using the TSL instruction

---

Test-and-Set

- Hardware guarantees that the instruction executes atomically on a CPU.
- Atomically: As an indivisible unit.
- The instruction can not stop half way through.

Test-and-Set on SMP

- It does not work without some extra hardware support

Test-and-Set on SMP

- A solution:
  - Hardware blocks all other CPUs from accessing the bus during the TSL instruction to prevent memory accesses by any other CPU.
  - TSL has mutually exclusive access to memory for duration of instruction.

Test-and-Set on SMP

- Test-and-Set is a busy-wait synchronisation primitive
- Called a spinlock
- Issue:
  - Lock contention leads to spinning on the lock
    - Spinning on a lock requires blocking the bus which slows all other CPUs down
    - Independent of whether other CPUs need a lock or not
    - Causes bus contention
- Caching does not help reduce bus contention
  - Either TSL still blocks the bus
  - Or TSL requires exclusive access to an entry in the local cache
    - Requires invalidation of same entry in other caches, and loading entry into local cache
    - Many CPUs performing TSL simply bounce a single exclusive entry between all caches using the bus
Reducing Bus Contention

• Read before TSL
  • Spin reading the lock variable waiting for it to change
  • When it does, use TSL to acquire the lock
• Allows lock to be shared read-only in all caches until its released
  • No bus traffic until actual release
• No race conditions, as acquisition is still with TSL.

start:
while (lock == 1);
r = TSL(lock)
if (r == 1)
goto start;

Comparing Simple Spinlocks

• Test and Set
  void lock (volatile lock_t *l) {
    while (test_and_set(l)) ;
  }
  
• Read before Test and Set
  void lock (volatile lock_t *l) {
    while (*l == BUSY || test_and_set(l)) ;
  }

Benchmark

for i = 1 .. 1,000,000 {
  lock(l)
  crit_section()
  unlock()
  compute()
}

• Compute chosen from uniform random distribution of mean 5 times critical section
• Measure elapsed time on Sequent Symmetry (20 CPU 30386, coherent write-back invalidate caches)

Results

• Test and set performs poorly once there is enough CPUs to cause contention for lock
  • Expected
• Read before Test and Set performs better
  • Performance less than expected
  • Still significant contention on lock when CPUs notice release and all attempt acquisition
• Critical section performance degenerates
  • Critical section requires bus traffic to modify shared structure
  • Lock holder competes with CPU that’s waiting as they test and set, so the lock holder is slower
  • Slower lock holder results in more contention
Spinning Locks versus Blocking Locks

Spinning versus Blocking and Switching

• Spinning (busy-waiting) on a lock makes no sense on a uniprocessor
  • The was no other running process to release the lock
  • Blocking and (eventually) switching to the lock holder is the only sensible option.

• On SMP systems, the decision to spin or block is not as clear.
  • The lock is held by another running CPU and will be freed without necessarily switching away from the requestor

Spinning versus Switching

• Blocking and switching
  • to another process takes time
    • Save context and restore another
    • Cache contains current process not new process
    • Adjusting the cache working set also takes time
    • TLB is similar to cache
  • Switching back when the lock is free encounters the same again
  • Spinning wastes CPU time directly

• Trade off
  • If lock is held for less time than the overhead of switching to and back
    • It’s more efficient to spin
  • Spinlocks expect critical sections to be short
  • No waiting for I/O within a spinlock
  • No nesting locks within a spinlock
Preemption and Spinlocks

- Critical sections synchronised via spinlocks are expected to be short
- Avoid other CPUs wasting cycles spinning

- What happens if the spinlock holder is preempted at end of holder’s timeslice
  - Mutual exclusion is still guaranteed
  - Other CPUs will spin until the holder is scheduled again!!!!!!

⇒ Spinlock implementations disable interrupts in addition to acquiring locks to avoid lock-holder preemption