Multiprocessor Systems

Chapter 8, 8.1

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
  - Assumes
    - 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

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 (a)
- Bus bandwidth becomes a bottleneck with more than just a few CPUs

Cache Consistency

- What happens if one CPU writes to address 0x1234 (and it is stored in its cache) and another CPU reads from the same address (and gets what is in its cache)?
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

Bus Based UMA

- To further scale the number processors, we give each processor private local memory
  - Keep private data local on off the shared memory bus
  - Bus bandwidth still becomes a bottleneck with many CPUs with shared data
  - Complicate application development
    - We have to partition between private and shared variables

Bus Based UMA

- With only a single shared bus, scalability is limited by the bus bandwidth of the single bus
  - Caching only helps so much
- Alternative bus architectures do exist on high-end machines

UMA Crossbar Switch

- Pro
  - Any CPU can access any available memory with less blocking
- Con
  - Number of switches required scales with $n^2$
    - 1000 CPUs need 1000000 switches

Summary

- Multiprocessors can
  - Increase computation power beyond that available from a single CPU
  - Share resources such as disk and memory
- However
  - Shared buses (bus bandwidth) limit scalability
    - Can be reduced via hardware design
    - Can be reduced by carefully crafted software behaviour
    - Good cache locality together with private data 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

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 have one processor thrashing, and the others with free memory
- No way to move free memory from one OS to another
- Consistency is an issue with independent disk buffer caches and potentially shared files

Master-Slave Multiprocessors

- OS (mostly) runs on a single fixed CPU
  - All OS tables, queues, buffers are present/manipulated on CPU 1
- User-level apps run on the other CPUs
  - And CPU 1 if there is spare CPU time
- All system calls are passed to CPU 1 for processing

Issue

- Master CPU can become the bottleneck
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

**Issue: Real concurrency in the kernel**
- Need carefully applied synchronisation primitives to avoid disaster
- Better cache locality
- The “big lock” becomes a bottleneck when in-kernel processing exceed what can be done on a single CPU

**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
- Only slight better solution than master slave

**Better alternative:** identify largely independent parts of the kernel and make each of them their own critical section
- Allows more parallelism in the kernel

**Better alternative:** identify largely independent parts of the kernel and make each of them their own critical section
- Allows more parallelism in the kernel

**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

**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
- Results in potentially complex lock ordering schemes that must be adhered to

**Example:**
- Given a “big lock” kernel, we divide the kernel into two independent parts with a lock each
- Leads to more subdivision, more locks, more complex lock acquisition rules
- Subdivision in practice is (in reality) making more code multithreaded

**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

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?

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

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

Entering and leaving a critical region using the TSL instruction
Test-and-Set

- Hardware guarantees that the instruction executes atomically.
  • Atomicity: 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 locks the bus during the TSL instruction to prevent memory accesses by any other CPU

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 bus locking which slows all other CPUs down
      - Independent of whether other CPUs need a lock or not
      - Causes bus contention

Test-and-Set on SMP

- Caching does not help reduce bus contention
  - Either TSL still locks 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.

```c
start:
while (lock == 1);
if (r == 1)
goto start;
```
Reducing Bus Contention

- If lock fails, use exponential back-off algorithm
  - Wait a longer and longer time before trying again
    - 2 instructions, 4, 8, 16 etc
    - Up to a maximum
    - High maximum gives lower bus contention
    - Low maximum gives more responsive locks

```
start:
  r = TSL(lock)
  if (r == 1) { exp_wait(); goto start; }
```

MCS Locks

- Each CPU enqueues its own private lock variable into a queue and spins on it
  - No contention
- On lock release, the releaser unlocks the next lock in the queue
  - Only have bus contention on actual unlock
  - No starvation (order of lock acquisitions defined by the list)

Other Hardware Provided SMP Synchronisation Primitives

- Atomic Add/Subtract
  - Can be used to implement counting semaphores
- Exchange
- Compare and Exchange
- Load linked; Store conditionally
  - Two separate instructions
    - Load value using load linked
    - Modify, and store using store conditionally
      - If value changed by another processor, or an interrupt occurred, then store conditionally failed
  - Can be used to implement all of the above primitives
  - Implemented without bus locking

Spinning versus Switching

- Remember spinning (busy-waiting) on a lock made little 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 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 blocking the requestor

Spinning versus Switching

- Blocking and switching
  - to another process takes time
    - Save context and restore another
    - Cache contains current process not new
    - 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
Spinning versus Switching

• The general approaches taken are
  – Spin forever
  – Spin for some period of time, if the lock is not acquired, block and switch
  • The spin time can be
    – Fixed (related to the switch overhead)
    – Dynamic
      » Based on previous observations of the lock acquisition time

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

Multiprocessor Scheduling

• Given X processes (or threads) and Y CPUs,
  – how do we allocate them to the CPUs

A Single Shared Ready Queue

• When a CPU goes idle, it take the highest priority process from the shared ready queue

Single Shared Ready Queue

• Pros
  – Simple
  – Automatic load balancing
• Cons
  – Lock contention on the ready queue can be a major bottleneck
    • Due to frequent scheduling or many CPUs or both
  – Not all CPUs are equal
    • The last CPU a process ran on is likely to have more related entries in the cache.

Affinity Scheduling

• Basic Idea
  – Try hard to run a process on the CPU it ran on last time
• One approach: Two-level scheduling
Two-level Scheduling

- Each CPU has its own ready queue
- Top-level algorithm assigns process to CPUs
  - Defines their affinity, and roughly balances the load
- The bottom-level scheduler:
  - Is the frequently invoked scheduler (e.g. on blocking on I/O, a lock, or exhausting a timeslice)
  - Runs on each CPU and selects from its own ready queue
  - Ensures affinity
  - If nothing is available from the local ready queue, it runs a process from another CPUs ready queue rather than go idle

Pros

- No lock contention on per-CPU ready queues in the (hopefully) common case
- Load balancing to avoid idle queues
- Automatic affinity to a single CPU for more cache friendly behaviour