MultiProcessors
Key Points

- What is a Chip Multiprocessor (CMP)?
- Why have we started building them?
- Why are they hard to use?
- What is deadlock?
- What is cache coherence?
- What is cache consistency?
Multiprocessors

• Put multiple processor chips in the same machine
• Until days of two-chip Pentium Pro machines,
  • very expensive
  • not used by many users
• Is this different than having multiple machines?
  • Processors share resources like DRAM, Net or Disk
  • Processors can communicate very fast (1 us vs. 1 ms)
• How do they communicate?
Multiprocessor Communication

- Message Passing
  - Simpler to implement; came first
  - Scalable Used today for very large supercomputers
  - Programming is generally not as easy
Multiprocessor Message Passing

• Simple ISA extension to send:

```
start_msg $3, $4 ; r3 is core #; r4 is message size
send_word $5 ; repeat for however many words
launch ; send into network
```

• Simple ISA extension to receive:

```
receive_word $5 ; stalls until word available
; message words are delivered in order
```

• “Fire and forget” semantics
• Works well for well-defined communication patterns
Multiprocessor Communication

- Shared Memory
  - Harder to implement; especially with caches
  - Not as scalable
    - Simplest designs scale to 8 cores
    - More advanced designs scale to ~100
  - Programming is somewhat easier
    - Programs share address space and communicate by reading/writing to the same DRAMs
    - Communication can be much more unstructured
Chip Multiprocessors (CMPS)

- Multiple processors on one die
- Aka multicore
- An easy way to spend xtrs
- Now common place
  - Laptops/desktops/game consoles/etc.
  - Less sophisticated users, all kinds of applications.
The Trouble With CMPs

- Amdahl’s law
  - $\text{Sot} = 1/(x/S + (1-x))$
- In order to double performance with a 2-way CMP
  - $S = 2$
  - $x = 1$
  - Usually, neither is achievable
Parallelism is Hard to Find

• To exploit CMP parallelism you need multiple processes or multiple “threads”
• Different Applications
  • Separate programs actually running (not sitting idle) on your computer at the same time.
  • Very common in servers
  • Less common in desktop/laptops
• Multiple Threads within one Application
  • Independent portions of your program that can run in parallel
  • Hard to change your program so that it can run in parallel
  • Most programs are not multi-threaded; at least not for the purpose of running faster
Parallel Programming is Hard

- **Difficulties**
  - Correctly identifying independent portions of complex programs
  - Sharing data between threads safely.
  - Synchronizing correctly (locks and messages)
  - Avoiding deadlock

- **Researchers still working on “good” solutions**
  - considered a “grand challenge” problem
  - no silver bullet; lots of different solutions that work well in particular contexts (e.g. OpenGL, MapReduce, ..)
Critical Sections and Locks

- A critical section is a piece of code that only one thread should be executing at a time.

```c
int shared_value = 0;
void IncrementSharedVariable()
{
    int t = shared_value + 1; // Line 1
    shared_value = t;         // line 2
}
```

- If two threads execute this code, we would expect the shared_value to go up by 2
- However, they could both execute line 1, and then both execute line 2 -- both would write back the same new value.

Instructions in the two threads can be interleaved in any way.
Critical Sections and Locks

• By adding a lock, we can ensure that only one thread executes the critical section at a time.

```c
int shared_value = 0;
lock shared_value_lock;
void IncrementSharedVariable()
{
    acquire(shared_value_lock);
    int t = shared_value + 1; // Line 1
    shared_value = t;         // line 2
    release(shared_value_lock);
}
```

• In this case we say shared_value_lock “protects” shared_value.
Locks are Hard

• The relationship between locks and the data they protect is not explicit in the source code and not enforced by the compiler
• In large systems, the programmers typically cannot tell you what the mapping is
• As a result, there are many bugs.
Locking Bug Example

```c
void Swap(int * a, lock * a_lock,
       int * b, lock * b_lock) {
    lock(a_lock);
    lock(b_lock);
    int t = a;
    a = b;
    b = t;
    unlock(a_lock);
    unlock(b_lock);
}
```

Thread 1

```
Swap(foo, foo_lock, 
    bar, bar_lock);
```

Thread 2

```
Swap(bar, bar_lock, 
    foo, foo_lock);
```

Thread 1 locks foo_lock, thread 2 locks bar_lock, both wait indefinitely for the other lock.
Finding, preventing, and fixing this kind of bug are all hard
Future of Parallel Programming

- Lots of New Approaches Being Tried Right Now
  - New languages, new architecture
  - New libraries
  - New paradigms
  - Revamped undergraduate programming courses
- But history has shown it’s very hard
  - There is probably not a good, general solution
  - We will make piecemeal progress
  - Many programs will hit an upper limit on speedup
  - CMPs just make your spyware run faster.
- Intel and Microsoft believe typical users can utilize up to about 8 cores effectively.
  - Your laptop will be there in 2-3 years.
Architectural Support for Multiprocessors

• Allowing multiple processors in the same system has a large impact on the memory system.
• How should processors see changes to memory that other processors make?
• How do we implement locks?
Shared Memory

- Multiple processors connected to a single, shared pool of DRAM
- If you don’t care about performance, this is relatively easy... but what about caches?
Uni-processor Caches

- Caches mean multiple copies of the same value
- In uniprocessors this is not a big problem
  - From the (single) processor’s perspective, the “freshest” version is always visible.
  - There is no way for the processor to circumvent the cache to see DRAM’s copy.
• With multiple caches, there can be many copies
• No one processor can see them all.
• Which one has the “right” value?

Local caches

Main Memory

Bus/arbiter

Store 0x1000
Read 0x1000
Store 0x1000

0x1000: A
0x1000: ??
0x1000: C
0x1000: B
Keeping Caches Synchronized

- We must make sure that all copies of a value in the system are up to date
  - We can update them
  - Or we can “invalidate” (i.e., destroy) them
- There should always be exactly one current value for an address
  - All processors should agree on what it is.
- We will enforce this by enforcing a total order on all load and store operations to an address and making sure that all processors observe the same ordering.
- This is called “Cache Coherence”
The Basics of Cache Coherence

- Every cache line (in each cache) is in one of 3 states
  - Shared -- There are multiple copies but they are all the same. Only reading is allowed
  - Owned (or Exclusive) -- This is the only cached copy of this data. Reading and write are allowed
  - Invalid -- This cache line does not contain valid data.
- There can be multiple sharers, but only one owner.
There is one copy of the state machine for each line in each coherence cache.
Invalid to Shared

Local caches

Shared 0x1000: Z

Read 0x1000

Bus/arbiter

Main Memory

0x1000: Z

Read 0x1000

Read 0x1000

Read 0x1000
Shared to Modified

Store A, 0x1000

Local caches

Shared 0x1000: Z

Shared 0x1000: Z

Shared 0x1000: Z

Shared 0x1000: Z

Bus/arbiter

Main Memory

0x1000: Z
Invalidate others

Store \(0x1000\)

Local caches

- Modified \(0x1000: A\)
- \(x\)
- \(x\)
- \(x\)

Bus/arbiter

Main Memory

- \(0x1000: Z\)
Modified to Shared

Store 0x1000

Read 0x1000

Local caches

Exclusive 0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Modified to Shared

Store 0x1000

Read 0x1000

Local caches

Shared 0x1000:A

Shared 0x1000:A

Main Memory

0x1000: A

Bus/arbiter
Shared to Modified

Store 0x1000
Read 0x1000
Store C, 0x1000

Local caches

Main Memory

Bus/arbiter

Shared 0x1000:A
Shared 0x1000:A

0x1000: A
Shared to Modified

Store 0x1000

Read 0x1000

Store C, 0x1000

Local caches

invalid
0x1000: A

invalid
0x1000: A

modified
0x1000: C

Bus/arbiter

Main Memory

0x1000: A
Coherence in Action

```
a = 0
Thread 1
while(1) {
    a++;
}
Thread 2
while(1) {
    print(a);
}

Sample outputs
1 1 1
2 1 2
3 1 5
4 1 8
5 100 3
6 100 5
7 100 2
8 100 4
possible? yes yes no
```
Coherence In The Real World

- Real coherence have more states
  - e.g. “Exclusive” -- I have the only copy, but it’s not modified
- Often don’t bother updating DRAM, just forward data from the current owner.
- If you want to learn more, take grad arch!
Cache Consistency

• If two operations occur in an order in one thread, we would like other threads to see the changes occur in the same order.
  • Example:

    Thread 0                        Thread 1
    A = 10;                          while(!A_is_valid);
    A_is_valid = true;               B = A;

  • We want B to end up with the value 10
  • Coherence does not give us this assurance, since the state machine only applies to a single cache line
  • This is called “cache consistency” or “the consistency model”
Simple Consistency

• The simplest consistency model is called “sequential consistency”
• In which all stores are immediately visible everywhere.

```
Thread 0
A = 10;
A_is_valid = true;

Thread 1
while(!A_is_valid);
B = A;
```

• If thread 1 sees the write to A_is_valid, it will also see the write to A.
What about this?

```c
a = b = 0

while(1) {
    a++;
    b++;
}

Thread 1

while(1) {
    print(a, b);
}

Thread 2

Sample outputs
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
1 1
2 2
3 1000
4 1000
5 1000
6 1000
7 1000
8 1000

possible under sequential consistency?

yes no
Consistency in the Real World

- Consistency is probably the most subtle aspect of computer architecture
- Sequential consistency is rarely implemented because it is too slow
  - Make all accesses visible everywhere, right away takes a long time
- Many machines use “relaxed” models.
  - All manner of non-intuitive things can happen
  - Special instructions to enforce sequential consistency when it’s needed
- Threading libraries (like pthreads) provide locking routines that use those special instructions to make locks work properly.