61C In the News

The New York Times
By STEVE LOHR
October 13, 2011

Dennis Ritchie, Trailblazer in Digital Era, Dies at 70

In the late 1960s and early '70s, working at Bell Labs, Mr. Ritchie made a pair of lasting contributions to computer science. **He was the principal designer of the C programming language** and co-developer of the Unix operating system, working closely with Ken Thompson *[UC Berkeley EECS BS '65, MS '66]* , his longtime Bell Labs collaborator.

Thompson (left) and Ritchie (center) receiving the National Medal of Technology from President Clinton in 1999.

[Kernighan and] Ritchie wrote a classic text, "The C Programming Language," also known as "K. & R."

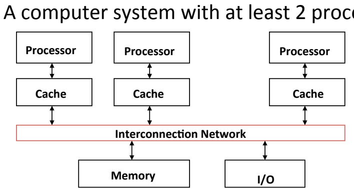
Mr. Ritchie joined Bell Labs in 1967, and soon began his fruitful collaboration with Mr. Thompson on both Unix and the C programming language. The pair represented the two different strands of the nascent discipline of computer science. Mr. Ritchie came to computing from math, while Mr. Thompson came from electrical engineering.

CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Lecture 21
Thread Level Parallelism II

Instructors:
Michael Franklin
Dan Garcia
http://inst.eecs.Berkeley.edu/~cs61c/Fa11

Review: Parallel Processing: Multiprocessor Systems (MIMD)

- MP - A computer system with at least 2 processors:



- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How many processors can be supported?

Shared Memory Multiprocessor (SMP)

- Q1 – Single address space shared by all processors/cores
- Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
  – Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time
- All multicore computers today are SMP

Example: Sum Reduction

- Sum 100,000 numbers on 100 processor SMP
  – Each processor has ID: 0 ≤ Pn ≤ 99
  – Partition 1000 numbers per processor
  – Initial summation on each processor [Phase I]

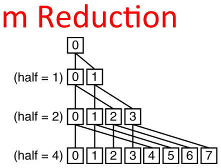
```
 sum[Pn] = 0;
   for (i = 1000*Pn;
       i < 1000*(Pn+1); i = i + 1)
     sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums [Phase II]
  – Reduction: divide and conquer
  – Half the processors add pairs, then quarter, …
  – Need to synchronize between reduction steps

Example: Sum Reduction

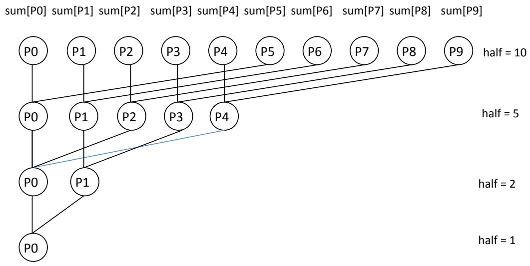
Second Phase:
After each processor has computed its "local" sum

This code runs simultaneously on each core

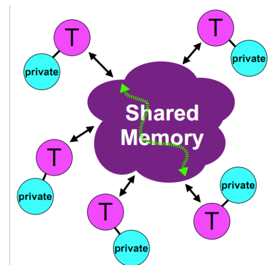



```
half = 100;
repeat
  synch();
  /*Proc 0 sums extra element if there is one */
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

An Example with 10 Processors



Memory Model for Multi-threading



- *All threads have access to the same, __globally shared__, memory*
- *Data can be shared or private*
- *Shared data is accessible by all threads*
- *Private data can only be accessed by the thread that owns it*
- *Data transfer is transparent to the programmer*
- *Synchronization takes place, but it is mostly implicit*

CAN BE SPECIFIED IN A LANGUAGE WITH MIMD SUPPORT – SUCH AS OPENMP

Peer Instruction

```
half = 100;
repeat
  synch();
  /*Proc 0 sums extra element if there is one */
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

What goes in Shared? What goes in Private?

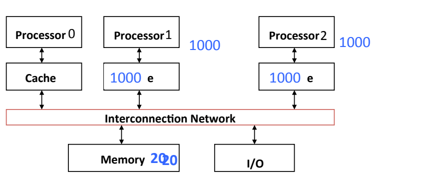
	half	sum	Pn
(a)	PRIVATE	PRIVATE	PRIVATE
(b)	PRIVATE	PRIVATE	SHARED
(c)	PRIVATE	SHARED	PRIVATE
(d)	SHARED	SHARED	PRIVATE
(e)	SHARED	SHARED	SHARED

Three Key Questions about Multiprocessors

- Q3 – How many processors can be supported?
- Key bottleneck in an SMP is the memory system
- Caches can effectively increase memory bandwidth/open the bottleneck
- But what happens to the memory being actively shared among the processors through the caches?

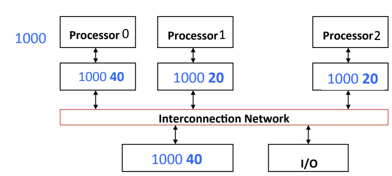
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



Processor 0 Write Invalidates Other Copies

Keeping Multiple Caches Coherent

- Architect's job: shared memory => keep cache values coherent
- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate all other copies
- Shared written result can "ping-pong" between caches

How Does HW Keep $ Coherent?

Each cache tracks state of each *block* in cache:

Shared: up-to-date data, not allowed to write
other caches may have a copy
copy in memory is also up-to-date

Modified: up-to-date, changed (dirty), OK to write
no other cache has a copy,
copy in memory is out-of-date

Invalid: Not really in the cache

2 Optional Performance Optimizations of Cache Coherency via new States

Exclusive: up-to-date data, OK to write
no other cache has a copy,
copy in memory up-to-date

- Avoids writing to memory if block replaced
- Supplies data on read instead of going to memory

Owner: up-to-date data, OK to write
other caches may have a copy (they must be in Shared state)
copy in memory not up-to-date

- Owner must supply data on read instead of going to memory

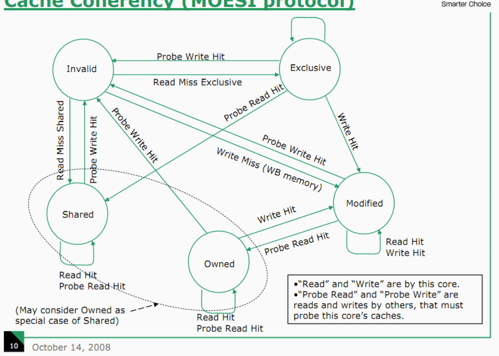
Common Cache Coherency Protocol: MOESI

- Each block in each cache is in one of the following states:
  - Modified (in cache)
  - Owned (in cache)
  - Exclusive (in cache)
  - Shared (in cache)
  - Invalid (not in cache)

	M	O	E	S	I
M	✗	✗	✗	✗	✓
O	✗	✗	✗	✓	✓
E	✗	✗	✗	✗	✓
S	✗	✓	✗	✓	✓
I	✓	✓	✓	✓	✓

Allowed states for a given cache block in any pair of caches

Cache Coherency (MOESI protocol)



Cache Coherency and Block Size

- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?
- Effect called *false sharing*
- How can you prevent it?

Threads

- *thread of execution*: smallest unit of processing scheduled by operating system
- On 1 processor, multithreading occurs *by time-division multiplexing*:
  - Processor switched between different threads
  - *Context switching* happens frequently enough user perceives threads as running at the same time
- On a multiprocessor, threads run at the same time, with each processor running a thread

Data Races and Synchronization

- Two memory accesses form a *data race* if from different threads to same location, and at least one is a write, and they occur one after another
- If there is a data race, result of program can vary depending on chance (which thread ran first?)
- Avoid data races by synchronizing writing and reading to get deterministic behavior
- Synchronization done by user-level routines that rely on hardware synchronization instructions

Lock and Unlock Synchronization

- Lock used to create region (*critical section*) where only one thread can operate
- Given shared memory, use memory location as synchronization point: *lock* or *semaphore*
- Thread reads lock to see if it must wait, or OK to go into critical section (and set to locked)
  - 0 => lock is free / open / unlocked / lock off
  - 1 => lock is set / closed / locked / lock on

```
Set the lock
Critical section
  (only one thread
  gets to execute
  this section of
  code at a time)
e.g., change shared
  variables
Unset the lock
```


Possible Lock/Unlock Implementation

- Lock (aka busy wait):

```
         addiu $t1,$zero,1   ; t1 = Locked value
Loop:    lw $t0,lock($s0)    ; load lock
         bne $t0,$zero,Loop  ; loop if locked
Lock:    sw $t1,lock($s0)    ; Unlocked, so lock
```

- Unlock:

```
         sw $zero,lock($s0)
```

- Any problems with this?

Student Roulette?

Possible Lock Problem

- Thread 1

```
      addiu $t1,$zero,1
Loop: lw $t0,lock($s0)


      bne $t0,$zero,Loop

Lock: sw $t1,lock($s0)
```

- Thread 2

```
      addiu $t1,$zero,1
Loop: lw $t0,lock($s0)


      bne $t0,$zero,Loop

Lock: sw $t1,lock($s0)
```

Time

Both threads think they have set the lock
Exclusive access not guaranteed!

Help! Hardware Synchronization

- Hardware support required to prevent interloper (either thread on other core or thread on same core) from changing the value
  - *Atomic* read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in MIPS

- Load linked: `ll rt,offset(rs)`
- Store conditional: `sc rt,offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt (clobbers register value being stored)
  - Fails if location has changed
    - Returns 0 in rt (clobbers register value being stored)
- Example: atomic swap (to test/set lock variable)
  Exchange contents of reg and mem: $s4 ←→ ($s1)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

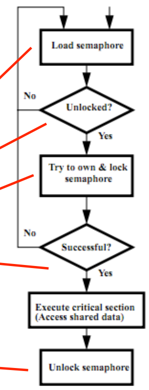

Test-and-Set in MIPS

- Example: MIPS sequence for implementing a T&S at ($s1)

```
Try: addiu $t0,$zero,1
     ll  $t1,0($s1)
     bne $t1,$zero,Try
     sc  $t0,0($s1)
     beq $t0,$zero,try
Locked:

     critical section

     sw $zero,0($s1)
```



And In Conclusion, …

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multiprocessor (Multicore) uses Shared Memory (single address space)
- Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern
- Synchronization via hardware primitives:
  - MIPS does it with Load Linked + Store Conditional
- Next Time: OpenMP as simple parallel extension to C