Advanced Operating Systems (CS 202)

Memory Consistency, Cache Coherence and Synchronization

(some cache coherence slides adapted from Ian Watson; some memory consistency slides from Sarita Adve)
Classic Example

Suppose we have to implement a function to handle withdrawals from a bank account:

```java
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

Now suppose that you and your father share a bank account with a balance of $1000

Then you each go to separate ATM machines and simultaneously withdraw $100 from the account
Interleaved Schedules

The problem is that the execution of the two threads can be interleaved:

```c
balance = get_balance(account);
balance = balance - amount;
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
put_balance(account, balance);
```

What is the balance of the account now?
How Interleaved Can It Get?

How contorted can the interleavings be?

- We'll assume that the only atomic operations are reads and writes of individual memory locations
- Some architectures don't even give you that!
- We'll assume that a context switch can occur at any time
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever

```c
.............. get_balance(account);
balance = get_balance(account);
balance = ...................................
balance = balance – amount;
balance = balance – amount;
put_balance(account, balance);
put_balance(account, balance);
```
Mutual Exclusion

- Mutual exclusion to synchronize access to shared resources
  - This allows us to have larger atomic blocks
  - What does atomic mean?

- Code that uses mutual called a critical section
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: sharing an ATM with others

- What requirements would you place on a critical section?
Using Locks

withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
Using Test-And-Set

Here is our lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

When will the while return? What is the value of held?
Overview

- Before we talk deeply about synchronization
  - Need to get an idea about the memory model in shared memory systems
  - Is synchronization only an issue in multi-processor systems?
- What is a shared memory processor (SMP)?
- Shared memory processors
  - Two primary architectures:
    - Bus-based/local network shared-memory machines (small-scale)
    - Directory-based shared-memory machines (large-scale)
Plan…

- Introduce and discuss cache coherence
- Discuss basic synchronization, up to MCS locks (from the paper we are reading)
- Introduce memory consistency and implications
- Is this an architecture class???
  - The same issues manifest in large scale distributed systems
Crash course on cache coherence
Bus-based Shared Memory Organization

Basic picture is simple :-

- CPU
- Cache

- CPU
- Cache

- CPU
- Cache

- Shared Memory

- Shared Bus
Organization

- Bus is usually simple physical connection (wires)
- Bus bandwidth limits no. of CPUs
- Could be multiple memory elements
- For now, assume that each CPU has only a single level of cache
Problem of Memory Coherence

- Assume just single level caches and main memory
- Processor writes to location in its cache
- Other caches may hold shared copies - these will be out of date
- Updating main memory alone is not enough
- What happens if two updates happen at (nearly) the same time?
  - Can two different processors see them out of order?
Example

Processor 1 reads X: obtains 24 from memory and caches it
Processor 2 reads X: obtains 24 from memory and caches it
Processor 1 writes 32 to X: its locally cached copy is updated
Processor 3 reads X: what value should it get?
    Memory and processor 2 think it is 24
    Processor 1 thinks it is 32

Notice that having write-through caches is not good enough
Cache Coherence

› Try to make the system behave as if there are no caches!

› How? Idea: Try to make every CPU know who has a copy of its cached data?
  › too complex!

› More practical:
  › Snoopy caches
    › Each CPU snoops memory bus
    › Looks for read/write activity concerned with data addresses which it has cached.
      › What does it do with them?
    › This assumes a bus structure where all communication can be seen by all.

› More scalable solution: ‘directory based’ coherence schemes
Snooping Protocols

- Write Invalidate
  - CPU with write operation sends invalidate message
  - Snooping caches invalidate their copy
  - CPU writes to its cached copy
    - Write through or write back?
  - Any shared read in other CPUs will now miss in cache and re-fetch new data.
Snooping Protocols

- Write Update
  - CPU with write updates its own copy
  - All snooping caches update their copy
- Note that in both schemes, problem of simultaneous writes is taken care of by bus arbitration - only one CPU can use the bus at any one time.
- Harder problem for arbitrary networks
Update or Invalidate?

- Which should we use?
- Bus bandwidth is a precious commodity in shared memory multi-processors
  - Contention/cache interrogation can lead to 10x or more drop in performance
  - (also important to minimize false sharing)
- Therefore, invalidate protocols used in most commercial SMPs
Cache Coherence summary

- Reads and writes are atomic
  - What does atomic mean?
    - As if there is no cache

- Some magic to make things work
  - Have performance implications
  - ...and therefore, have implications on performance of programs
So, lets try our hand at some synchronization
What is synchronization?

- Making sure that concurrent activities don’t access shared data in inconsistent ways

```java
int i = 0; // shared

Thread A                  Thread B
 i = i + 1;                i = i - 1;

What is in i?
```
What are the sources of concurrency?

- Multiple user-space processes
  - On multiple CPUs
- Device interrupts
- Workqueues
- Tasklets
- Timers
Pitfalls in scull

Race condition: result of uncontrolled access to shared data

```c
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos]) {
        goto out;
    }
}
```

Scull is the Simple Character Utility for Locality Loading (an example device driver from the Linux Device Driver book)
Pitfalls in scull

 Race condition: result of uncontrolled access to shared data

```c
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos]) {
        goto out;
    }
}
```
Pitfalls in scull

- *Race condition*: result of uncontrolled access to shared data

```c
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos]) {
        goto out;
    }
}
```
Pitfalls in scull

- **Race condition**: result of uncontrolled access to shared data

```c
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos]) {
        goto out;
    }
}
```

Memory leak
Synchronization primitives

- **Lock/Mutex**
  - To protect a shared variable, surround it with a lock (critical region)
  - Only one thread can get the lock at a time
  - Provides mutual exclusion

- **Shared locks**
  - More than one thread allowed (hmm…)

- **Others?** Yes, including Barriers (discussed in the paper)
Synchronization primitives (cont’d)

- Lock based
  - Blocking (e.g., semaphores, futexes, completions)
  - Non-blocking (e.g., spin-lock, …)
    - Sometimes we have to use spinlocks
- Lock free (or partially lock free 😊)
  - Atomic instructions
  - seqLocks
  - RCU
  - Transactions
How about locks?

- **Lock(L):**
  - If(L==0)
    - L=1;
  - else
    - while(L==1);
    - //wait
    - go back;

- **Unlock(L):**
  - L=0;

Can we do this just with atomic reads and writes?

Yes but not easy—Decker’s algorithm
Easier to use read-modify-update atomic instructions
Naïve implementation of spinlock

- **Lock(L):**
  
  ```c
  While(test_and_set(L));
  //we have the lock!
  //eat, dance and be merry
  ```

- **Unlock(L)**
  
  ```c
  L=0;
  ```
Why naïve?

- Works? Yes, but not used in practice
- Contention
  - Think about the cache coherence protocol
  - Set in test and set is a write operation
    - Has to go to memory
    - A lot of cache coherence traffic
    - Unnecessary unless the lock has been released
    - Imagine if many threads are waiting to get the lock
- Fairness/starvation
Better implementation: Spin on read

- Assumption: We have cache coherence
  - Not all are: e.g., Intel SCC

- Lock(L):
  ```c
  while(L==locked); //wait
  if(test_and_set(L)==locked) go back;
  ```

- Still a lot of chattering when there is an unlock
  - Spin lock with backoff
Bakery Algorithm

```c
struct lock {
    int next_ticket;
    int now_serving;
}

// Acquire_lock:
int my_ticket = fetch_and_inc(L->next_ticket);
while(L->new_serving!=my_ticket); //wait
//Eat, Dance and me merry!

// Release_lock:
L->now_serving++;
```

Comments? Fairness? Efficiency/cache coherence?
Anderson Lock (Array lock)

- Problem with bakery algorithm:
  - All threads listening to next_serving
    - A lot of cache coherence chatter
  - But only one will actually acquire the lock
  - Can we have each thread wait on a different variable to reduce chatter?
Anderson’s Lock

- We have an array (actually circular queue) of variables
  - Each variable can indicate either lock available or waiting for lock
    - Only one location has lock available

**Lock(L):**

```
my_place = fetch_and_inc (queuelast);
while (flags[myplace mod N] == must_wait);
```

**Unlock(L)**

```
flags[myplace mod N] = must_wait;
flags[mypalce+1 mod N] = available;
```

Fair and not noisy – compare to spin-on-read and bakery algorithm
Any negative side effects?
Concurrency and Memory Consistency

References:
- *A primer on memory consistency and cache coherence*, Sorin, Hill and wood, 2011 (chapters 3 and 4)
- *Memory Models: A Case for Rethinking Parallel Languages and Hardware*, Adve and Boehm, 2010
Memory Consistency

- Formal specification of memory semantics
- Guarantees as to how shared memory will behave on systems with multiple processors
- Ordering of reads and writes
- Essential for programmer (OS writer!) to understand
Why Bother?

- Memory consistency models affect *everything*
  - Programmability
  - Performance
  - Portability
- Model must be defined at all levels
- Programmers and system designers care
Uniprocessor Systems

- Memory operations occur:
  - One at a time
  - In program order
- Read returns value of last write
  - Only matters if location is the same or dependent
  - Many possible optimizations

- Intuitive!
How does a core reorder? (1)

- Store-store reordering:
  - Non-FIFO write buffer

- Load-load or load-store/store-load reordering:
  - Out of order execution

- Should the hardware prevent any of this behavior?
## Multiprocessor: Example

### TABLE 3.1: Should r2 Always be Set to NEW?

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: Store data = NEW;</td>
<td>L1: Load r1 = flag;</td>
<td>/* Initially, data = 0 &amp; flag ≠ SET */</td>
</tr>
<tr>
<td>S2: Store flag = SET;</td>
<td>B1: if (r1 ≠ SET) goto L1;</td>
<td>/* L1 &amp; B1 may repeat many times */</td>
</tr>
<tr>
<td></td>
<td>L2: Load r2 = data;</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 3.2: One Possible Execution of Program in Table 3.1.

<table>
<thead>
<tr>
<th>cycle</th>
<th>Core C1</th>
<th>Core C2</th>
<th>Coherence state of data</th>
<th>Coherence state of flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S2: Store flag=SET</td>
<td>L1: Load r1=flag</td>
<td>read-only for C2</td>
<td>read-write for C1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>L2: Load r2=data</td>
<td>read-only for C2</td>
<td>read-only for C2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>read-only for C2</td>
<td>read-only for C2</td>
</tr>
<tr>
<td>4</td>
<td>S1: Store data=NEW</td>
<td></td>
<td>read-write for C1</td>
<td>read-only for C2</td>
</tr>
</tbody>
</table>

- S2 and S1 reordered
- Why? How?
Table 3.3: Can Both $r_1$ and $r_2$ be Set to 0?

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: $x = \text{NEW}$;</td>
<td>S2: $y = \text{NEW}$;</td>
<td>/* Initially, $x = 0$ &amp; $y = 0$*/</td>
</tr>
<tr>
<td>L1: $r_1 = y$;</td>
<td>L2: $r_2 = x$;</td>
<td></td>
</tr>
</tbody>
</table>
Sequential Consistency

- The result of any execution is the same as if all operations were executed on a single processor.
- Operations on each processor occur in the sequence specified by the executing program.
One execution sequence

**TABLE 3.1**: Should r2 Always be Set to NEW?

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: Store data = NEW;</td>
<td>L1: Load r1 = flag;</td>
<td>/* Initially, data = 0 &amp; flag ≠ SET */</td>
</tr>
<tr>
<td>S2: Store flag = SET;</td>
<td>B1: if (r1 ≠ SET) goto L1;</td>
<td>/* L1 &amp; B1 may repeat many times */</td>
</tr>
<tr>
<td></td>
<td>L2: Load r2 = data;</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 3.1**: A Sequentially Consistent Execution of Table 3.1’s Program.
program order (<p) of Core C1

S1: \( x = \text{NEW}; /* \text{NEW} */ \)
L1: \( r1 = y; /* 0 */ \)

memory order (<m)

S2: \( y = \text{NEW}; /* \text{NEW} */ \)
L2: \( r2 = x; /* \text{NEW} */ \)

program order (<p) of Core C2

Outcome: \((r1, r2) = (0, \text{NEW})\)

(a) SC Execution 1

(b) SC Execution 2

S1: \( x = \text{NEW}; /* \text{NEW} */ \)
L1: \( r1 = y; /* \text{NEW} */ \)

S2: \( y = \text{NEW}; /* \text{NEW} */ \)
L2: \( r2 = x; /* 0 */ \)

Outcome: \((r1, r2) = (\text{NEW}, 0)\)

(c) SC Execution 3

(d) NOT an SC Execution
S.C. Disadvantages

- Difficult to implement!
- Huge lost potential for optimizations
  - Hardware (cache) and software (compiler)
  - Be conservative: err on the safe side
  - Major performance hit
Relaxed Consistency

- **Program Order** relaxations \((\text{different locations})\)
  - \(W \rightarrow R; \quad W \rightarrow W; \quad R \rightarrow R/W\)
- **Write Atomicity** relaxations
  - Read returns another processor’s Write early
- Combined relaxations
  - Read your own Write \((\text{okay for S.C.})\)
- **Safety Net** – available synchronization operations
- *Note: assume one thread per core*
Synchronization is broken!

How can we solve this problem?

Answer: Memory Barrier/Fence

- A special compiler or CPU instruction that enforces an ordering constraint
- Compiler: `asm volatile ("" ::: "memory");`
- CPU: `mfence/lfence`