Advanced Operating Systems (CS 202)

Memory Consistency, Cache Coherence and Synchronization

Jan, 27, 2017

(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:

```plaintext
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:

```
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;
}

- Why is the “return” outside the critical section? Is this ok?
- What happens when a third thread calls acquire?
Stepping back

• What does the OS need to support?
  – And why? Isn’t this an application/programming problem?

• Synchronization is hard – why?

• Synchronization can be a performance problem – why?

• Other semantics than mutual exclusion possible.
Implementing locks

- Software implementations possible
  - You should have seen Dekker’s algorithm and possibly Peterson’s algorithm
  - They are difficult to get right
  - They make assumptions on the system that may no longer hold
    - (e.g., memory consistency as we will see shortly)

- Most systems offer hardware support
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         |       |       CPU         |       |       CPU         |
|       | Cache     |       |       | Cache     |       |       | Cache     |
+-------------------+       +-------------------+       +-------------------+
|                           |       |                           |
| Shared Bus               |       | Shared Bus               |
|                           |       |                           |
|       +-------------------+       |       +-------------------+
|       Shared Memory      |       |       Shared Memory      |
```

13
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
- Other organizations (e.g., with a network) have NUMA issues
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?
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
Implementation Issues

• In both schemes, knowing if a cached value is not shared (copy in another cache) can avoid sending any messages.

• Invalidate description assumed that a cache value update was written through to memory. If we used a ‘copy back’ scheme other processors could re-fetch old value on a cache miss.

• We need a protocol to handle all this.
MESI – locally initiated accesses

Invalid

Modified

Shared

Exclusive

Mem Read

Read Miss(sh)

Read Miss(ex)

Invalidate

Write Hit

Write Hit

Write Hit

= bus transaction
MESI – remotely initiated accesses

- **Invalid**
  - RWITM
  - Mem Read
  - Invalidate

- **Shared**
  - RWITM
  - Mem Read

- **Modified**
  - RWITM

- **Exclusive**
  - RWITM

= copy back
Both together

Image credit: Wiki page on MESI
By Jugones55 - Own work, GFDL, https://commons.wikimedia.org/w/index.php?curid=7136764
MESI notes

• There are other protocols and minor variations (particularly to do with write miss)

• Normal ‘write back’ when cache line is evicted is done if line state is M

• Multi-level caches
  – If caches are inclusive, only the lowest level cache needs to snoop on the bus
    • Most modern CPUs have inclusive caches
    • But they don’t perform as well as non-inclusive caches
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
Directory Schemes

• Snoopy schemes do not scale because they rely on broadcast

• Directory-based schemes allow scaling.
  - avoid broadcasts by keeping track of all PEs caching a memory block, and then using point-to-point messages to maintain coherence
  - they allow the flexibility to use any scalable point-to-point network
Basic Scheme (Censier & Feautrier)

- Assume "k" processors.
- With each cache-block in memory: k presence-bits, and 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

Read from main memory by PE-i:
- If dirty-bit is OFF then { read from main memory; turn p[i] ON; }
- if dirty-bit is ON then { recall line from dirty PE (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to PE-i; }

Write to main memory:
- If dirty-bit OFF then { send invalidations to all PEs caching that block; turn dirty-bit ON; turn P[i] ON; ... }
Key Issues

• Scaling of memory and directory bandwidth
  - Can not have main memory or directory memory centralized
  - Need a distributed memory and directory structure

• Directory memory requirements do not scale well
  - Number of presence bits grows with number of PEs
  - Many ways to get around this problem
    • limited pointer schemes of many flavors

• Industry standard
  - SCI: Scalable Coherent Interface
SO, LET'S TRY OUR HAND AT SOME SYNCHRONIZATION
What is synchronization?

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

• int i = 0; // shared
  
  Thread A
  i = i+1;
  
  Thread B
  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 (!dp->data[s_pos]) {
    dp->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dp->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;
    }
}
```
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, futexases, completions)
  - Non-blocking (e.g., spin-lock, …)
    • Sometimes we have to use spinlocks

• **Lock free (or partially lock free 😊)**
  - Atomic instructions
  - seqLocks
  - RCU
  - Transactions (next time)
Example

P1
Modify struct (A)
flag = 1

P2
Wait for modification
Use struct (A)

Is it possible to use only read/write instructions to achieve this?

code:

Modify (A)
flag = 1
while (flag == 0);
use(A)
flag = 0
But how about locks?

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

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

Check and lock are not atomic!

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

• **Unlock(L)**
  
  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?
• Each node has:
  struct node {
    bool got_it;
    Next; //successor
  }

Lock(L, me)
join(L); //use fetch-n-store
while(got_it == 0);

Unlock(L, me)
remove me from L
signal successor
(setting got it to 0)
Race condition!

```plaintext
type qnode = record
    next : ^qnode
    locked : Boolean
end

// parameter I, below, points to a qnode record allocated
// (in an enclosing scope) in shared memory locally-accessible
// to the invoking processor

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor := ^qnode := fetch_and_store (L, I)
    if predecessor != nil  // queue was non-empty
        I->locked := true
        predecessor->next := I
    repeat while I->locked  // spin

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil  // no known successor
        if compare_and_swap (L, I, nil)
            return
        // compare_and_swap returns true iff it swapped
    repeat while I->next = nil  // spin
    I->next->locked := false
```

• What if there is a new joiner when the last element is removing itself

48
Figure 1: Pictorial example of MCS locking protocol in the presence of competition.