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:

  Execution sequence seen by CPU

  ```
  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

```
............... 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

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
Cache

CPU
Cache

CPU
Cache

Shared Bus

Shared Memory
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?
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, 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 (!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;
    }
}
```
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 (next time)
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?
Check and lock are not atomic!
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):
  ```
  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):

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

Unlock(L):

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