Hardware and Software Synchronization
Advanced Computer Architecture
COMP 140
Thursday June 26, 2014
Synchronization

- In a multithread system, synchronization is extremely important in a memory system — the key hardware capability is an **uninterruptible instruction or instruction sequence capable of atomically retrieving and changing a value**.

- Basic problem:
  - If two concurrent threads are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior.
Once a hardware synchronization mechanism exists, software synchronization mechanisms are then constructed using the capability.

“Locks” are used to create “mutual exclusion” and to implement more complex synchronization mechanisms.
Synchronization

- Threads cooperate in multithreaded programs
  - To share resources, access shared data structures
    - Threads accessing a memory cache in a Web server
  - To coordinate their execution
    - One thread executes relative to another (recall ping-pong)
- For correctness, we need to control this cooperation
  - Threads *interleave executions arbitrarily* and at different rates
  - Scheduling is not under program control
- Cooperation is controlled using synchronization
  - Restrict the possible interleavings
- We’ll discuss in terms of threads, also applies to processes
Synchronization 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 significant other share a bank account with a balance of $1000.
- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account.
Synchronization Example

- We’ll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run in the same bank process:

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

- What’s the problem with this implementation?
  - Think about potential schedules of these two threads
Synchronization Example

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

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

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

- What is the balance of the account now?
- This is known as a race condition
- Context switch
  - Each thread is “racing” to `put_balance()` before the other
Mutual Exclusion

- One way to ensure who wins the race is to only let one thread “compete”; this is called mutual exclusion
- Code that uses mutual exclusion to synchronize its execution is 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

```java
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    return balance;
}
```
Critical Section Requirements

1. Mutual exclusion
   • If one thread is in the critical section, then no other is

2. Progress
   • If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section

3. Bounded waiting (no starvation)
   • If some thread T is waiting on the critical section, then T will eventually enter the critical section

4. No assumptions on performance
   • Requirements must be met with any number of CPUs with arbitrary relative speeds
Locks

• One way to implement critical sections is to “lock the door” on the way in, and unlock it again on the way out.

• A lock is an object in memory providing two operations:
  • acquire(): before entering the critical section
  • release(): after leaving a critical section

• Threads pair calls to acquire() and release():
  • Between acquire()/release(), the thread holds the lock
  • acquire() does not return until any previous holder releases
  • What can happen if the calls are not paired?
Using Locks

withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}

acquire(lock);
balance = get_balance(account);
balance = balance - amount;
acquire(lock);
put_balance(account, balance);
release(lock);
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
release(lock);

• What happens when green tries to acquire the lock?
• Why is the “return” outside the critical section? Is this ok?
• What happens when a third thread calls acquire?
First Attempt: Spin Locks

- How do we implement locks? Here is one attempt:

```c
struct lock {
    int held = 0;
}

void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}

void release (lock) {
    lock->held = 0;
}
```

- This is called a spinlock because a thread spins waiting for the lock to be released.
- Does this work?
First Attempt: Spin Locks

- Does this work? Nope. Two independent threads may both notice that a lock has been released and thereby acquire it.

```c
struct lock {
    int held = 0;
};

void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}

void release (lock) {
    lock->held = 0;
}
```

A context switch can occur here, causing a race condition.
Can we take turns?

- What if we took turns (assuming only two threads)

```c
struct lock {
    int held = 0;
}

void acquire (lock) {
    while (lock->turn != this_thread);
    lock->held = 1;
}

void release (lock) {
    lock->turn = other_thread;
}
```

- Does this work? Why not?
Can we take turns? Nope. Can we declare intent?

- A thread doesn’t know if the other thread is ready.
- What if we wait until the other thread isn’t interested:

```
struct lock {
    int interested[2] = [FALSE, FALSE];
};

void acquire (lock) {
    lock->interested[this_thread] = TRUE;
    while (lock->interested[other_thread]);
}

void release (lock) {
    lock->interested[this_thread] = FALSE;
}
```

- Now does it work?
Peterson’s Algorithm

- Take turns only if somebody else is interested; otherwise just go!

```c
struct lock {
    int turn = 0;
    int interested[2] = [FALSE, FALSE];
}

void acquire (lock) {
    lock->interested[this_thread] = TRUE;
    turn = other_thread;
    while (lock->interested[other_thread] && turn==other_thread);
}

void release (lock) {
    lock->interested[this_thread] = FALSE;
}
```

- Finally works, but only if we know who else is playing.
Hardware Support to the Rescue!

- If we build atomic operations into the hardware, then we can enforce single-thread critical sections, enforcing **mutual exclusion**:
  - Two processes can never be in the critical section at the same time.
  - We now have code that executes “all or nothing”
Hardware Synchronization

- **Basic building blocks:**
  - Atomic exchange
    - Swaps register with memory location
  - Test-and-set
    - Sets under condition
  - Fetch-and-increment
    - Reads original value from memory and increments it in memory
    - Requires memory read and write in uninterruptable instruction
  - Load linked/store conditional
    - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails
Atomic Exchange Synchronization

- Atomic exchange interchanges a value in a register for a value in memory
- Assume a simple lock, where value 0 is used to indicate that the lock is free, and 1 is used to indicate that the lock is unavailable.
- The processor thread tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock. The value that is returned is 1 if some other processor had already claimed access, and 0 otherwise (success).
- If successful, the value is changed to 1, preventing any competing exchange from retrieving 0.
Test and Set Synchronization

- Test and Set semantics (similar to Atomic Exchange):
  - Record the old value \textit{and}
  - Set the value to indicate available \textit{and}
  - Return the old value
- Hardware automatically executes test and set atomically:

```c
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

- When executing test-and-set on “flag”
  - What is value of flag afterwards if it was initially False? True?
  - What is the return result if flag was initially False? True?
Test and Set Synchronization

- Simple 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?

- The problem with this spin-lock is that it is wasteful:
  - On single processor machines, when a thread is spinning, the thread holding the lock isn’t making progress!
    - The lock holder gave up the CPU in the first place by either sleeping/yielding, or involuntarily.
  - Not as bad on multiprocessor machines, but still need to be used cautiously: only for a short period of time.
Spin Locks

- With no cache coherence, we could build a spin lock by keeping the lock variable in memory.
- A processor could continually try to acquire the lock using an atomic operation (e.g., atomic exchange) and test whether the exchange returned the lock as free.
- To release the lock, the processor just stores the value 0 to the lock.
- MIPS spinlock whose address is in $1:

  ```
  DADDUI $2,$0,#1 ; put 1 into $2
  
  lockit: EXCH $2,0($1) ; atomic exchange
  BNEZ $2,lockit ; already locked?
  ```
Spin Locks: coherence

- With coherence, we can use the coherence mechanism to cache the locks and maintain them.
- Caching has two advantages:
  - Spinning happens on cached values, not main memory
  - There is often locality in lock accesses: the processor that used the lock last will use it again. The lock resides in that processor’s cache, reducing the time to acquire it.
- We do need a change to our spin procedure
  - we don’t want multiple processors to generate a write while spinning — most would be write misses
  - each processor would be trying to obtain the lock variable in an exclusive state
Spin Locks: coherence

- Modify our spin lock procedure so it spins by doing reads on a local copy of the lock until it successfully sees that the lock is available.
- Then, attempt to acquire the lock by doing a swap.
  - First, read the lock variable to test its state
  - Keep reading and testing until the value of the read indicates the lock is unlocked.
- Race against all other processes that want the lock, by attempting a swap.
- The winner will see 0 (from the swap) and the losers will see 1. When the winner finishes its critical section, it stores a 0 in the lock, and the race is on again.
Spin Locks: coherence

- Spin lock code (0 is unlocked, 1 is locked):

  ```
  lockit:   LD       R2,0(R1) ; load of lock
            BNEZ     R2,lockit ; not available-spin
            DADDUI   R2,R0,#1 ; load locked value
            EXCH     R2,0(R1) ; swap
            BNEZ     R2,lockit ; branch if lock wasn’t 0
  ```

- How does this use the coherence mechanism?
Spin Locks: coherence

- Once the processor with the lock stores a 0 into the lock, all other caches are invalidated, and much fetch the new value to update their copy of the lock.
- One cache gets the copy of the unlocked value (0) first and performs the swap.
- When the cache miss of other processors is satisfied, they find that the variable is already locked, so they go back to testing and spinning.
Spin Locks: coherence

<table>
<thead>
<tr>
<th>Step</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>Coherence state of lock at end of step</th>
<th>Bus/directory activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Has lock</td>
<td>Begins spin, testing if lock = 0</td>
<td>Begins spin, testing if lock = 0</td>
<td>Shared</td>
<td>Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.</td>
</tr>
<tr>
<td>2</td>
<td>Set lock to 0</td>
<td>(Invalidate received)</td>
<td>(Invalidate received)</td>
<td>Exclusive (P0)</td>
<td>Write invalidate of lock variable from P0.</td>
</tr>
<tr>
<td>3</td>
<td>Cache miss</td>
<td>Cache miss</td>
<td>Shared</td>
<td>Bus/directory services P2 cache miss; write-back from P0; state shared.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(Waits while bus/directory busy)</td>
<td>Lock = 0 test succeeds</td>
<td>Shared</td>
<td>Cache miss for P2 satisfied</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Lock = 0</td>
<td>Executes swap, gets cache miss</td>
<td>Shared</td>
<td>Cache miss for P1 satisfied</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Executes swap, gets cache miss</td>
<td>Completes swap: returns 0 and sets lock = 1</td>
<td>Exclusive (P2)</td>
<td>Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Swap completes and returns 1, and sets lock = 1</td>
<td>Enter critical section</td>
<td>Exclusive (P1)</td>
<td>Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Spins, testing if lock = 0</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 5.24 Cache coherence steps and bus traffic for three processors, P0, P1, and P2.* This figure assumes write invalidate coherence. P0 starts with the lock (step 1), and the value of the lock is 1 (i.e., locked); it is initially exclusive and owned by P0 before step 1 begins. P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3 to 5). P2 wins and enters the critical section (steps 6 and 7), while P1’s attempt fails so it starts spin waiting (steps 7 and 8). In a real system, these events will take many more than 8 clock ticks, since acquiring the bus and replying to misses take much longer. Once step 8 is reached, the process can repeat with P2, eventually getting exclusive access and setting the lock to 0.
Models of Memory Consistency

Processor 1:
A=0
...
A=1
if (B==0) …

Processor 2:
B=0
...
B=1
if (A==0) …

• Should be impossible for both if-statements to be evaluated as true
  • Delayed write invalidate?

• Sequential consistency:
  • Result of execution should be the same as long as:
    • Accesses on each processor were kept in order
    • Accesses on different processors were arbitrarily interleaved
Memory consistency

• To implement, delay completion of all memory accesses until all invalidations caused by the access are completed
• Reduces performance!

• Example: Suppose we have a processor where a write miss takes 50 cycles to establish ownership, 10 cycles to issue each invalidate after ownership is established, and 80 cycles for an invalidate to complete and be acknowledged once it is issued. Assuming that four other processors share a cache block, how long does a write miss stall the writing processor if the processor is sequentially consistent? Assume that the invalidates must be explicitly acknowledged before the coherence controller knows they are completed. Suppose we could continue executing after obtaining ownership for the write miss without waiting for the invalidates; how long would the write take?
Memory Consistency

- **Answer:** When we wait for invalidates, each write takes the sum of the ownership time plus the time to complete the invalidates. Since the invalidates can overlap, we need only worry about the last one, which starts $10 + 10 + 10 + 10 = 40$ cycles after ownership is established. Hence, the total time for the write is $50 + 40 + 80 = 170$ cycles. In comparison, the ownership time is only 50 cycles. With appropriate write buffer implementations, it is even possible to continue before ownership is established.

- **Alternatives:**
  - Program-enforced synchronization to force write on processor to occur before read on the other processor
  - Requires synchronization object for A and another for B
  - “Unlock” after write
  - “Lock” after read
Relaxed Consistency Models

- Rules:
  - $X \rightarrow Y$
    - Operation $X$ must complete before operation $Y$ is done
  - Sequential consistency requires:
    - $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow R$, $W \rightarrow W$

- Relax $W \rightarrow R$
  - “Total store ordering” (“processor consistency”). Retains order among writes, so many programs operate just fine.

- Relax $W \rightarrow W$
  - “Partial store order”

- Relax $R \rightarrow W$ and $R \rightarrow R$
  - “Weak ordering” and “release consistency”

If the programmer doesn’t understand the model under which the processor behaves — use expert-built standard synchronization libraries!
## Multicore Characteristics Summary

<table>
<thead>
<tr>
<th>Feature</th>
<th>AMD Opteron 8439</th>
<th>IBM Power 7</th>
<th>Intel Xenon 7560</th>
<th>Sun T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>904 M</td>
<td>1200 M</td>
<td>2300 M</td>
<td>500 M</td>
</tr>
<tr>
<td>Power (nominal)</td>
<td>137 W</td>
<td>140 W</td>
<td>130 W</td>
<td>95 W</td>
</tr>
<tr>
<td>Max. cores/chip</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Multithreading</td>
<td>No</td>
<td>SMT</td>
<td>SMT</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Threads/core</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Instruction issue/clock</td>
<td>3 from one thread</td>
<td>6 from one thread</td>
<td>4 from one thread</td>
<td>2 from 2 threads</td>
</tr>
<tr>
<td>Clock rate</td>
<td>2.8 GHz</td>
<td>4.1 GHz</td>
<td>2.7 GHz</td>
<td>1.6 GHz</td>
</tr>
<tr>
<td>Outermost cache</td>
<td>L3; 6 MB; shared</td>
<td>L3; 32 MB (using embedded DRAM); shared or private/core</td>
<td>L3; 24 MB; shared</td>
<td>L2; 4 MB; shared</td>
</tr>
<tr>
<td>Inclusion</td>
<td>No, although L2 is superset of L1</td>
<td>Yes, L3 superset</td>
<td>Yes, L3 superset</td>
<td>Yes</td>
</tr>
<tr>
<td>Multicore coherence protocol</td>
<td>MOESI</td>
<td>Extended MESI with behavioral and locality hints (13-state protocol)</td>
<td>MESIF</td>
<td>MOESI</td>
</tr>
<tr>
<td>Multicore coherence implementation</td>
<td>Snooping</td>
<td>Directory at L3</td>
<td>Directory at L3</td>
<td>Directory at L2</td>
</tr>
<tr>
<td>Extended coherence support</td>
<td>Up to 8 processor chips can be connected with HyperTransport in a ring, using directory or snooping. System is NUMA.</td>
<td>Up to 32 processor chips can be connected with the SMP links. Dynamic distributed directory structure. Memory access is symmetric outside of an 8-core chip.</td>
<td>Up to 8 processor cores can be implemented via Quickpath Interconnect. Support for directories with external logic.</td>
<td>Implemented via four coherence links per processor that can be used to snoop. Up to two chips directly connect, and up to four connect using external ASICs.</td>
</tr>
</tbody>
</table>

**Figure 5.27** Summary of the characteristics of four recent high-end multicore processors (2010 releases) designed for servers. The table includes the highest core count versions of these processors; there are versions with lower core counts and higher clock rates for several of these processors. The L3 in the IBM Power7 can be all shared or partitioned into faster private regions dedicated to individual cores. We include only single-chip implementations of multicores.
Peterson’s Algorithm: https://en.wikipedia.org/wiki/Peterson's_algorithm
Synchronization with Locks: http://cseweb.ucsd.edu/classes/fa05/cse120/lectures/120-l5.pdf