Lect. 8: Synchronization

- **Synchronization** is necessary to ensure that operations in a parallel program happen in the correct order
  - Condition synchronization
  - Mutual exclusion

- Different primitives are used at different levels of abstraction
  - High-level (e.g., monitors, parallel sections and loops): supported in languages themselves or language extensions (e.g., Java threads, OpenMP)
  - Middle-level (e.g., locks, barriers, and condition variables): supported in libraries (e.g., POSIX threads)
  - Low-level (e.g., compare&swap, test&set, load-link & store-conditional, transactional memory): supported in hardware

- Higher level primitives can be constructed from lower level ones
- Things to consider: deadlock, livelock, starvation
Example: Sync. in Java Threads

- **Synchronized Methods**
  - Concurrent calls to the method on the same object have to be serialized
  - All data modified during one call to the method becomes atomically visible to all calls to other methods of the object
  - E.g.:

    ```java
    public class SynchronizedCounter {
        private int c = 0;

        public synchronized void increment() {
            c++;
        }
    }
    
    SynchronizedCounter myCounter;
    
    - Can be implemented with locks
Example: Sync. in OpenMP

- Doall loops
  - Iterations of the loop can be executed concurrently
  - After the loop, all processors have to wait and a single one continues with the following code
  - All data modified during the loop is visible after the loop
  - E.g.:

```c
#pragma omp parallel for
private(i,s) shared (A,B)
schedule(static)
for (i=0; i<N; i++) {
    s = ...

    A[i] = B[i] + s;
}
```

- Can be implemented with barrier
Example: Sync. in POSIX Threads

- Locks
  - Only one thread can own the lock at any given time
  - Unlocking makes all the modified data visible to all threads and locking forces the thread to obtain fresh copies of all data
  - E.g.:

```c
pthread_mutex_t mylock;

pthread_mutex_init(&mylock, NULL);
pthread_mutex_lock(&mylock);
Count++;

pthread_mutex_unlock(&mylock);
```

- Can be implemented with hardware atomic RMW (e.g. test&set)
Example: Building Locks from Ld/St?

- E.g., Peterson’s algorithm

Processor 0

```c
int A, B, C;
int flag[2], turn;

flag[0]=0; flag[1]=0;
turn = 0;

/* lock */
flag[0] = 1; turn = 1;
While(flag[1]&&turn==1);

/* unlock */
flag[0] = 0;
```

Processor 1

```c
/* lock */
flag[1] = 1; turn = 0;
While(flag[0]&&turn==0);

/* unlock */
flag[1] = 0;
```
Example: Building Locks from Ld/St?

- Requires SC.
  - Relaxed models need to use fences.
- Works for only 2 processors
- A general N processor solution (for e.g. Bakery algorithms) requires O(N) flag variables and it also slow.
Hardware Primitives

- Hardware job is to provide atomic memory operations, which involves both processors and the memory subsystem
- Implemented in the IS, but usually encapsulated in library function calls by manufacturers
- At a minimum, hardware must provide an atomic swap
- Examples:
  - Compare&Swap (e.g., Sun Sparc): if value in memory is equal to value in register Ra then swap memory value with the value in Rb
    \[
    \text{CAS } (R1), R2, R3 \ ; \text{ if } (\text{MEM}[R1]==R2) \\
    \text{MEM}[R1]<->R3;
    \]
- Can implement more complex conditions for synchronization
- Requires comparison operation in memory or must block memory location until processor is done with comparison
- In addition the swap must be performed atomically
Hardware Primitives

- Examples:
  - Fetch&Increment (e.g., Intel x86) (in general Fetch&Op): increment the value in memory and return the old value in register

    \[
    \text{lock; ADD (R1),R2 ;MEM[R1]=MEM[R1]+R2}
    \]

    - Less flexible than Compare&Swap
    - Requires arithmetic operation in memory or must block memory location (or bus) until processor is done with addition (e.g., x86)

  - Swap (test-and-set): swap the values in memory and in a register
    - Less flexible of all
    - Does not require comparison or arithmetic operation in memory
    - But swap must be performed atomically
Implementing RMWs

- Need to guarantee atomicity of R and W
- Lock the bus
  - Early implementations
  - Slow (impacts other processors too)
- Cache line locking
  - Obtain exclusive access by doing a Read exclusive
  - Deny coherence requests until W completes
Example: Test&Set

```c
int lock(int *mylock) {

    int value;

    value = test&set(mylock, 1);
    if (value)
        return FALSE;
    else
        return TRUE;
}

void unlock(int *mylock) {
    *mylock = 0;
    return;
}
```
What If the Lock is Taken?

- **Spin-wait lock**
  
  ```c
  while (!lock(&mylock));
  ...
  unlock(&mylock);
  ```
  
  - Each call to lock invokes the hardware primitive, which involves an expensive memory operation and takes up network bandwidth

- **Spin-wait on cache: Test-and-Test&Set**
  
  ```c
  while (TRUE) {
    if (!lock(&mylock))
      while (!mylock);
    else break;
  }
  ...
  unlock(&mylock);
  ```
  
  - Spin on cached value using normal load and rely on coherence protocol
  
  - Still, all processors race to memory, and clash, once the lock is released
What If the Lock is Taken?

- **Software solution:** Blocking locks and Backoff

```c
while (TRUE) {
    if (lock(&mylock)) wait (time);
    else break;
}
...
unlock(&mylock);
```

- Wait can be implemented in the application itself (backoff) or by calling the OS to be put to sleep (blocking)
- The waiting time is usually increased exponentially with the number of retries
- Similar to the backoff mechanism adopted in the Ethernet protocol
LL/SC

- **Load-link and Store-conditional**
  - Implement atomic memory operation as two operations
  - **Load-link (LL):**
    - Registers the intention to acquire the lock
    - Returns the present value of the lock
  - **Store-conditional (SC):**
    - Only stores the new value if no other processor attempted a store between our previous LL and now
    - Returns 1 if it succeeds and 0 if it fails
  - Relies on the coherence mechanism to detect conflicting SC’s
  - All operation is done locally at the cache controllers or directory, no need for complex blocking operation in memory
  - Introduced in the MIPS processor, now also used in PowerPC and ARM
Another Hardware Primitive

- Load-link and Store-conditional operation

![Diagram showing the Coherence substrate with RESERVE operations in P0 and P1.]
Another Hardware Primitive

- Load-link and Store-conditional operation
Another Hardware Primitive

- Load-link and Store-conditional operation

LL 0xA

RESERVE

Coherence substrate

LL completes
Another Hardware Primitive

- Load-link and Store-conditional operation

Coherence substrate

LL completes

SC 0xA

RESERVE 0xA

RESERVE

P0

L1

P1

L1

Coherence substrate

RESERVE

P0

L1

RESERVE

P1

L1

Coherence substrate
Another Hardware Primitive

- Load-link and Store-conditional operation

Coherence substrate
Another Hardware Primitive

- Load-link and Store-conditional operation

Diagram:

- SC 0xA
  - RESERVE
  - P0
  - L1
  - L1
  - SC succeeds

- LL 0xA
  - RESERVE
  - P0
  - L1
  - L1
  - LL completes

Coherence substrate
Another Hardware Primitive

- Load-link and Store-conditional operation

SC 0xA

Coherence substrate

LL 0xA

Coherence substrate

RESERVE

RESERVE

RESERVE

L1

L1

L1

P0

P0

P1

P1
Another Hardware Primitive

- Load-link and Store-conditional operation

SC 0xA

RESERVE P0 L1

RESERVE P1 L1

LL completes

Coherence substrate

LL 0xA

RESERVE P0 L1

RESERVE P1 L1

LL completes

Coherence substrate
Another Hardware Primitive

- Load-link and Store-conditional operation
Another Hardware Primitive

- Load-link and Store-conditional operation

---

Coherence substrate

---

Coherence substrate

---
Another Hardware Primitive

- Load-link and Store-conditional operation
Another Hardware Primitive

- Load-link and Store-conditional operation
Building Locks with LL/SC

- E.g., spin-wait with attempted swap

```
try:   MOV   R2, lock
      LL    R1, location ; value of lock loaded
      BNZ   R1, try       ; try again if lock taken
      SC    R2, location  ; try to store conditionally
      BEQZ  R2, try       ; branch if SC failed
      RET
```

Advantages:
- Lesser complexity on coherence system – does not suffer from deadlocks
- Failing SC does not send invalidates
- Lends itself naturally to test-and-test&set like implementation
Transactional Memory (TM)

- Coarse-grain locking is easy but limits concurrency
- Fine-grain locking is efficient but hard to get right
- Can we get the performance of the latter with the programmability of the former?

- TM
  - Atomic read-modify-writes for sections of code.
  - First proposed in the context of database transactions
  - First proposed as replacement for locks by Herlihy and Moss in 1993
  - Intel Haswell architecture has TM implementation
HashMap

- Given key returns value if found
- Not thread-safe

```java
get(Object key)
{
    int id = hash(key);
    HashEntry e = buckets[id];
    while(e!=NULL)
    {
        if (key == e.key) return e.value;
        e = e.next;
    }
}
```
Making HashMap thread safe

- Grab a mutex before entering get, put
  - Coarse-grain locking
  - Easy to program
  - Limits concurrency

- Redesign using per-bucket locks
  - Fine-grain locking
  - Error prone and complex

- Use TM
  - Simply enclose get and put within Atomic
  - System ensures atomicity.
How system ensures Atomicity?

- Optimistic concurrency control
  - First proposed by Kung in 1981.
  - If two transactions conflict (if one modifies location read or modified by other), then one of the transactions aborts and rolls back.

- TM implementation must provide
  - Versioning: the ability to recover in case transaction does not succeed
    - Eager
    - Lazy
  - Conflict detection
    - Eager
    - Lazy
How system ensures Atomicity?

- **Software TM**
  - Versioning and Conflict detection performed in software by instrumenting loads and stores

- **Hardware TM**
  - Versioning: Buffer the state in local caches
  - Conflict detection: using cache coherence protocol
An Alternative Hdw. Approach

- Locks have a relatively large overhead and, thus, are suitable for guarding relatively large amounts of data
- Some algorithms need to exchange only a small number of words each time
- Also, consumer thread must wait for all data guarded by a lock to be ready before it can begin work
- Better approach for fine-grain synchronization: **Full/Empty Bits**
  - Associate one bit with every memory word (1.5% overhead for 64bit words)
  - Augment the behavior of load/store
    - Load: if word is empty then trap to OS (to wait)
      otherwise, return value and set bit to empty
    - Store: if word is full then trap to OS (to deal with error)
      otherwise, store the new value, set bit to full, and release any threads pending on the word (with OS help)
    - Reset: set the bit to empty
  - Good for producer-consumer type of communication/synchronization
References and Further Reading

- **Transactional memory**
  

- **A commercial machine with Full/Empty bits:**
  

- **Performance evaluations of synchronization for shared-memory:**
  
  