# CS5460: Operating Systems

## Lecture 9: Implementing Synchronization

***(Chapter 6)***

# Multiprocessor Memory Models

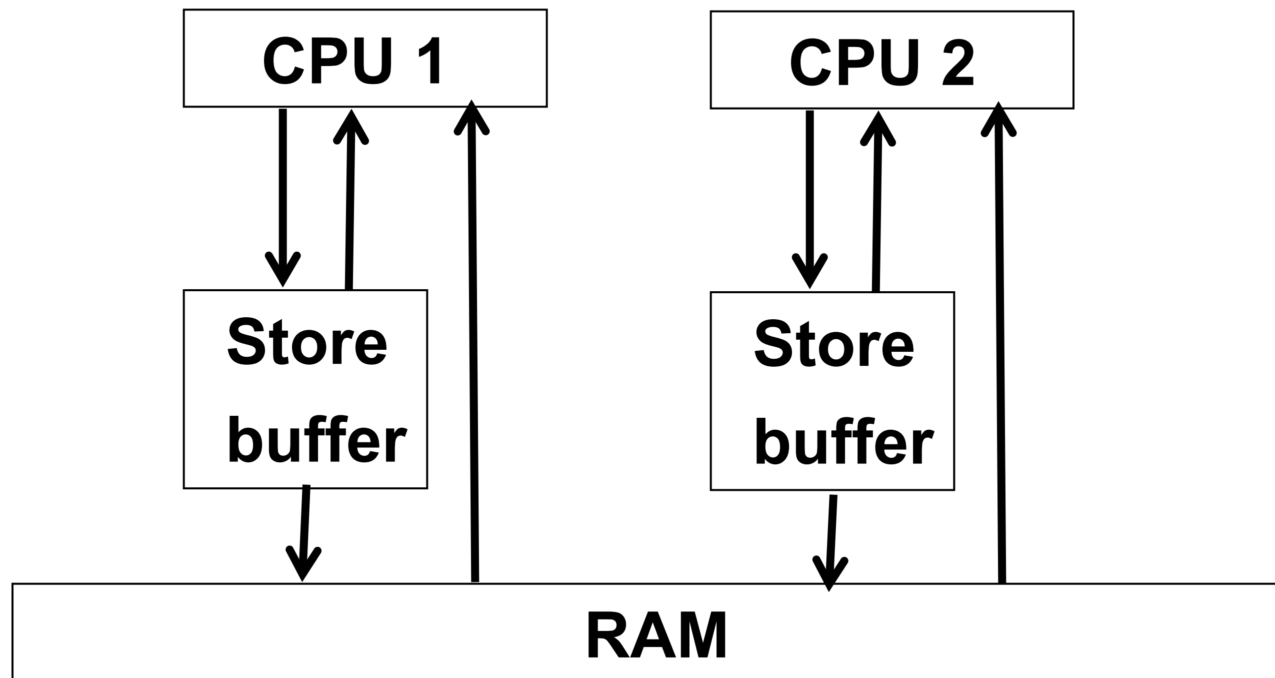
- **Uniprocessor memory is simple**
  - Every load from a location retrieves the last value stored to that location
  - Caches are transparent
  - All processes / threads see the same view of memory
- **The straightforward multiprocessor version of this memory model is "sequential consistency":**
  - "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program. Operations performed by each processor occur in the specified order."
  - This is Lamport's definition

# Multiprocessor Memory Models

- Real multiprocessors do not provide sequential consistency
  - Loads may be reordered after loads (IA64, Alpha)
  - Stores may be reordered after stores (IA64, Alpha)
  - Loads may be reordered after stores (IA64, Alpha)
  - Stores may be reordered after loads (many, including x86 / x64)
- Even on a uniprocessor, compiler can reorder memory accesses

# x86 / x86-64 Memory Model: TSO

- TSO: "Total store ordering"

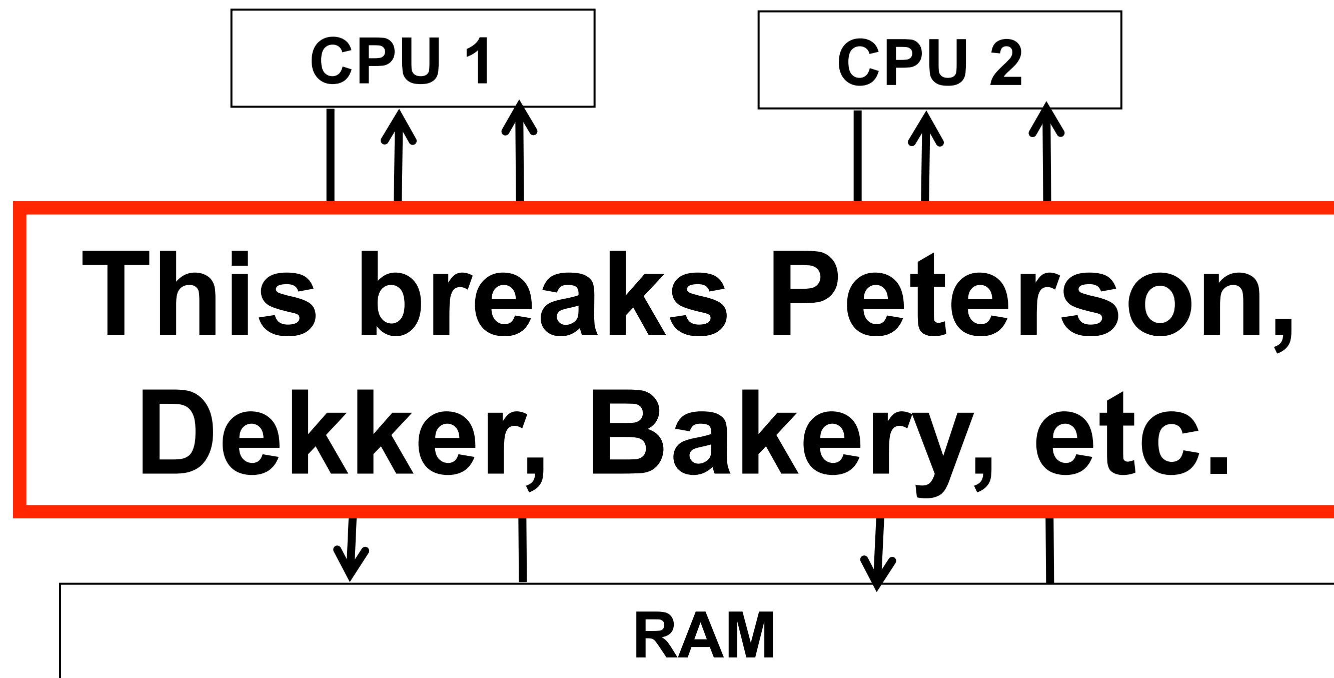




THE UNIVERSITY OF UTAH

# x86 / x86-64 Memory Model: TSO

- TSO: “Total store ordering”



# Weak Memory Example

- (This is the same as the code I sent out last week)
- Initially x and y are 0
- Now run in parallel:
  - CPU 0: x=1 ; print y
  - CPU 1: y=1 ; print x
- What might be printed on a sequentially consistent machine?
- What might be printed on a TSO machine?

# Memory Fences

- The x86 “mfence” instruction is your weapon against having your programs broken by TSO
  - Loads and stores cannot be moved before or after the mfence instruction
  - Basically you can think about it as flushing the store buffer and preventing the pipeline from reordering around the fence
- mfence is not cheap
  - But see “sfence” and “lfence” which are weaker (and faster) than mfence

# Weak Memory Example

- Initially x and y are 0
- Now run in parallel:
  - CPU 0: x=1 ; mfence ; print y
  - CPU 1: y=1 ; mfence ; print x
- What might be printed on a sequentially consistent machine?
- What might be printed on a TSO machine?

# Some good news for programmers…

- If your multithreaded code is free of data races, you don't have to worry about the memory model
  - Execution will be "sequentially consistent"
  - Acquire/release of locks include fences
- "Free of data races" means every byte of memory is
  - Not shared between threads
  - Shared, but in a read-only fashion
  - Shared, but consistently protected by locks
- Your goal is to always write programs that are free of data races!
  - Programs you write for this course will have data races – but these should be a rare exception

# If You Do Write Data Races

- **Accidental data race → Always a serious bug**
  - Means you don't understand your code
- **Deliberate data race →**
  - Executions no longer sequentially consistent
  - Dealing with the memory system and compiler optimizations is now your problem
  - Always ask: Why am I writing racy code?

# Writing Correct Racy Code

1. **Mark all racing variables as "volatile"**
   - `volatile int x[10];`
   - This keeps the compiler from optimizing away and reordering memory references
2. **Use memory fences, atomic instructions, etc. as needed**
   - These keep the memory system from reordering operations and breaking your code

# Dekker Sync. Algorithm

```
static int f0, f1, turn;

void lock_p0 (void) {
  f0 = 1;
  while (f1) {
    if (turn != 0) {
      f0 = false;
      while (turn != 0) { }
      f0 = true;
    }
  }
}
```

# Dekker Sync. Algorithm

```
static int f0, f1, turn;

void lock_p0 (void) {
  f0 = 1;
  while (f1) {
    if (turn != 0) {
      f0 = false;
      while (turn != 0) { }
      f0 = true;
    }
  }
}
```

**GCC turns this into:**

```
lock_p0:
    movl   $1, f0(%rip)
    ret
```

THE UNIVERSITY OF UTAH

# Reminder

- **For any mutual exclusion implementation, we want to be sure it guarantees:**
  - **Cannot allow multiple processes in critical section at the same time (mutual exclusion)**
  - **Ensure progress (lack of deadlock)**
  - **Ensure fairness (lack of livelock)**
- **We also want to know what invariants hold over the lock's data structures**

# Implementing Mutual Exclusion

- Option 1: Build on atomicity of loads and stores
  - Peterson, Bakery, Dekker, etc.
  - Loads and stores are weak, tedious to work with
  - Portable solutions do not exist on modern processors
- Option 2: Build on more powerful atomic primitives
  - Disable interrupts → keep scheduler from performing context switch at "unfortunate" time
  - Atomic synchronization instructions
    - » Many processors have some form of atomic: Load-Op-Store
    - » Also: Load-linked → Store-conditional (ARM, PPC, MIPS, Alpha)
- Common synchronization primitives:
  - Semaphores and locks (similar)
  - Barriers
  - Condition variables
  - Monitors

# Lock by Disabling Interrupts V.1

```
class Lock {
  public:
    void Acquire();
    void Release();
}

Lock::Lock {
}
```

```
Lock::Acquire() {
  disable interrupts;
}
Lock::Release() {
  enable interrupts;
}
```

THE UNIVERSITY OF UTAH

# Lock by Disabling Interrupts V.2

```
class Lock {
  public:
    void Acquire();
    void Release();
  private:
    int locked;
    Queue Q;
}

Lock::Lock {
  locked ← 0;    // Lock free
  Q ← 0;         // Queue empty
}
```

```
Lock::Acquire(T:Thread) {
  disable interrupts;
  if (locked) {
    add T to Q;
    T → Sleep();
  }
  locked ← 1;
  enable interrupts;
}
Lock::Release() {
  disable interrupts;
  if (Q not empty) {
    remove T from Q;
    put T on readyQ;
  }
  else locked ← 0;
  enable interrupts;
}
```

# Lock by Disabling Interrupts V.2

```
class Lock {
  public:
    void Acquire();
    void Release();
  private:
    int locked;
    Queue Q;
}

Lock::Lock {
  locked ← 0;   // Lock free
  Q ← 0;        // Queue empty
}
```

```
Lock::Acquire(T:Thread) {
  disable interrupts;
  if (locked) {
    add T to Q;
    T → Sleep();
  }
  locked ← 1;
  enable interrupts;
}
Lock::Release() {
  disable interrupts;
  if (Q not empty) {
    remove T from Q;
    put T on readyQ;
  }
  else locked ← 0;
  enable interrupts;
}
```

When do you enable ints.?

THE UNIVERSITY OF UTAH

# Blocking vs. Not Blocking?

- Option 1: Spinlock
- Option 2: Yielding spinlock
- Option 3: Blocking locks
- Option 4: Hybrid solution – spin for a little while and then block

- How do we choose among these options
  - On a uniprocessor?
  - On a multiprocessor?

# Problems With Disabling Interrupts

- Disabling interrupts for long is always bad
  - Can result in lost interrupts and dropped data
  - The actual max value depends on what you're doing
- Disabling interrupts (briefly!) is heavily used on uniprocessors
- But what about multiprocessors?
  - Disabling interrupts on just the local processor is not very helpful
    - » Unless we know that all processes are running on the local processor
  - Disabling interrupts on all processors is expensive
  - In practice, multiprocessor synchronization is usually done differently

# Hardware Synchronization Ops

- `test-and-set(loc, t)`
  - Atomically read original value and replace it with "t"
- `compare-and-swap(loc, a, b)`
  - Atomically: if (loc == a) { loc = b; }
- `fetch-and-add(loc, n)`
  - Atomically read the value at loc and replace it with its value incremented by n
- `load-linked / store-conditional`
  - load-linked : loads value from specified address
  - store-conditional : if no other thread has touched value → store, else return error
  - Typically used in a loop that does "read-modify-write"
  - Loop checks to see if read-modify-write sequence was interrupted

# Using Test&Set ("Spinlock")

- test&set(loc, value)
  - Atomically tests old value and replaces with new value
- Acquire()
  - If free, what happens?
  - If locked, what happens?
  - If more than one at a time trying to acquire, what happens?
- Busy waiting
  - While testing lock, process runs in tight loop
  - What issues arise?

```
class Lock {
  public:
    void Acquire(), Release();
  private:
    int locked;
}

Lock::Lock() { locked ← 0;}

Lock::Acquire() {
  // Spin atomically until free
  while (test&set(locked,1));
}

Lock::Release() { locked ← 0;}
```

THE UNIVERSITY OF UTAH

# Using Test&Set (Improved V.1)

- **test&set(loc, value)**
  - Atomically tests old value and replaces with new value
- **Acquire()**
  - If free, what happens?
  - If locked, what happens?
  - If more than one at a time trying to acquire, what happens?
- **Busy waiting**
  - What is new?

```
class Lock {
  public:
    void Acquire(), Release();
  private:
    int locked;
}

Lock::Lock() { locked ← 0;}

Lock::Acquire() {
  for (;;) {
    // Spin non-atomically first;
    while(locked);
    if (!test&set(locked,1)) break;
  }
}

Lock::Release() { locked ← 0;}
```

# Using Test&Set (Improved V.2)

- **test&set(loc, value)**
  - Atomically tests old value and replaces with new value
- **Acquire()**
  - If free, what happens?
  - If locked, what happens?
  - If more than one at a time trying to acquire, what happens?
- **Queueing**
  - What changes?

```
class Lock {
  public:  void Acquire(), Release();
  private: int locked, guard;
}
Lock::Lock() { locked ← guard ← 0;}
Lock::Acquire(t:Thread) {
  while(test&set(guard,1));
  if (locked) {
    put T to sleep, guard ← 0; }
  else { locked ← 1; guard ← 0; }
}
Lock::Release() {
  while(test&set(guard,1));
  if (Q not empty) { wake up T; }
  else {locked ← 0;}
  guard ← 0;
}
```

THE UNIVERSITY OF UTAH

# Compare&Swap

- **Cmp&swap(loc, a, b)**
  - Atomically tests if loc contains a; if so, stores b into loc
  - Returns old value from loc
- **Acquire()**
  - If free, what happens?
  - If locked, what happens?
  - If more than one at a time trying to acquire, what happens?
- **Busy waiting**
  - Can improve by:
    - » Backing off if compare fails
    - » Introducing wait queue

```
class Lock {
  public:
    void Acquire(), Release();
  private:
    int locked;
}

Lock::Lock() { locked ← 0;}

Lock:: Acquire() {
  // Spin atomically until free
  while (cmp&swap(locked,0,1));
}

Lock::Release() { locked ← 0;}
```

THE UNIVERSITY OF UTAH

# *Fetch&Add*

- **Fetch&add(loc, val)**
  - Atomically reads `loc`, adds `val` to it, and writes new value back
- **Acquire()**
  - If free, what happens?
  - If locked, what happens?
  - If more than one at a time trying to acquire, what happens?
- **Busy waiting**
  - Can improve by:
    - » Backing off if compare fails
    - » Introducing wait queue

```
class Lock {
  public:
    void Acquire(), Release();
  private:
    int counter = 0;
    turn = 0;
}

Lock:: Acquire() {
  // Spin atomically until free
  int me;
  me = fetch&add(counter,1);
  while (me != turn);
}

Lock::Release() { fetch&add(turn,1);}
```

# Load-linked, Store-conditional

- `LL(loc)`
  - Loads `loc` and puts address in special register for snoop HW
- `SC(loc)`
  - Conditionally stores to `loc`
  - Must be same addr as last `LL( )`
  - Succeeds iff `loc` not modified since `LL( )`
  - Fails if modified, or there has been a context switch
- Typically used in “try-retry” loops
- Optimistic concurrency control
  - Optimize for common case
  - Locking → pessimistic c.c.

```
// increment counter w/ LL-SC (MIPS)
                          // r1: &counter
try: ll   r2,(r1) // LL counter to r2
     addi r3,r2,1 // r3 ← r2+1
     sc   r3,(r1) // SC new counter
     beq  r3,0,try // Test if success
```

```
void arch_spin_lock (arch_spinlock_t *lock) {
  unsigned long tmp;
  u32 newval;
  arch_spinlock_t lockval;
  __asm__ __volatile__(
"1:     ldrex   %0, [%3]\n"
"       add     %1, %0, %4\n"
"       strex   %2, %1, [%3]\n"
"       teq     %2, #0\n"
"       bne     1b"
    : "=&r" (lockval), "=&r" (newval), "=&r" (tmp)
    : "r" (&lock->slock), "I" (1 << TICKET_SHIFT)
    : "cc");
  while (lockval.tickets.next != lockval.tickets.owner) {
    wfe();
    lockval.tickets.owner = ACCESS_ONCE(lock->tickets.owner);
  }
  smp_mb();
}
```

# Performance Issues

- **Spinlocks (busy waiting)**
  - Ties up processor while thread continually tests value
  - On an SMP, test&set( ) on a shared variable crosses network
  - Some possible optimizations:
    - » Test and test&set( ) : Use non-atomic instruction to do initial test
    - » Back off (yield/sleep) between each probe (constant? variable?)
    - » Implement per-lock queues → extra overhead, but fair
  - When is spinning preferred?
- **Consider where lock variable resides:**
  - Spinning on local cache versus spinning over the interconnect
- **Consider granularity of locking**
  - Lock entire data structure versus lock individual elements

# Summary

- The multiprocessor memory model is not your friend
  - Write race-free programs whenever possible
- Common hardware synchronization primitives
  - `test-and-set, compare-and-swap, fetch-and-add, LL/SC`
- Implementing spinlocks
- Performance issues:
  - Spinning versus queuing
  - Cache-aware synchronization policies
  - Scheduling-aware synchronization policies

# *Questions?*