Mutual Exclusion and Synchronization Mechanisms

Instructor: Prof. Cristina Silvano, email: silvano@elet.polimi.it
Teaching Assistant: Dr. Sotirios Xydis, email: xydis@elet.polimi.it
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

May 2013
Basic Concepts
Difference of Mutual Exclusion and Synchronization

- **Mutual Exclusion**
  - How do I protect a shared resource from concurrent access

- **Synchronization**
  - How to coordinate running threads/processes running in parallel, e.g. to wait for events to happen

- **Synchronization mechanisms build upon mutual exclusion primitives**
The problem of synchronization (1)

- The problem: if multiple processors operate on shared variables, final result must anyway be deterministic - by repeating execution with the same data but with potentially different order of access to variables, results must not change. It is thus necessary to synchronize accesses by the different processors.

- In abstract terms, a purely SW solution is possible; actually, HW supports are adopted.
The problem of synchronization (2)

- Typical aspect:
  - A process must be able to complete a given operation on a given data structure without being interrupted by any other process - i.e., it must be able to complete a “critical section” of the program in non-interruptible mode (it acquires the data structure and nobody else can intervene on the same until it “releases” it at the end of the critical section).
The problem of synchronization

- Synchronization mechanisms: typically built with user-level SW procedures using synchronization instructions provided by HW.
  - In small multiprocessors and in low contention situations: main HW capability is a non-interruptible instruction/instruction sequence capable of reading and modifying a value in atomic manner; SW mechanisms are then built using this capacity (e.g., spin locks)
Basic hardware Primitive

- A basic operation to build synchronization operations must allow reading and modifying a memory location in atomic fashion - i.e., so that the overall operation once initiated will complete without being interrupted by any other one. (There are many alternative versions of primitives of such type).

- There are many solutions for mutual exclusion: lock-unlock pairs, synchronization on point-to-point events by means of flags, global synchronization on events by means of barriers.
Main Components of a synchronization event

Three main components:

- **Method of acquisition** by which a process attempts to acquire the right to synchronization (i.e., to enter the critical section or to proceed beyond a synchronization event).

- **Waiting algorithm**: the method by which the process waits for synchronization to become available.

- **Method of release**, by which the process allows other processes to proceed beyond a synchronization event.
Synchronization event - Waiting Algorithm

- Choice of the waiting algorithm: independent of the synchronization type. Two main alternatives:
  - **busy-waiting** (the process attempting synchronization cycles waiting for a variable to change its value: release of the synchronization event by another process changes the variable’s value and allows the waiting process to proceed)
  - **Blocking** (the waiting process simply blocks, releasing the processor if it must wait; it will be waked up at the time of release).

- **Blocking solution** has greater overhead, because it calls the operating system, but it frees the processor for other activities;
- **Busy-waiting** avoids the cost of suspension but consumes processor time and bandwidth. (Busy-waiting better for short waiting times).
Mutual Exclusion
Mutual exclusion

- Mutual exclusion operations (lock-unlock): fast in situations of low contention for the lock, inefficient when contention is high.

- What is the use of a lock, L?
- Example: two processes Pj, Pk run on two nodes A and B; both modify the same data structure D, the two modifications do not imply precedence but must be executed each in atomic fashion, from beginning to end.
Basic Hardware Primitives

Check L; if free, acquire it and continue, otherwise wait
Operate on D
Release L

Check L; if free, acquire it and continue, otherwise wait
Operate on D
Release L
Typical operation for building synchronization operations:

Atomic exchange - Swaps value in a register with one in memory.

1. To implement a simple lock: if value is 0 lock is free, if value is 1 lock is not available.
2. Processor attempts blocking the lock by exchanging the value 1 (in a register) with memory location L corresponding to the lock;
3. Value returned by exchange instruction: 1 if some other node had already requested (and acquired) access, 0 otherwise. In the second case, value is set to 1 by the atomic exchange instruction, granting that no other competing exchange will read 0, and the processor has acquired the lock.
A more modern solution: pair of instructions - the second instruction returns a value from which it can be deduced whether the pair has been executed as if it were an atomic operation. Pair includes:

1. Load linked (or “load locked”)
2. Store conditional

The two are used in sequence.
Load Linked - Store Conditional (LL-SC) Synch. Primitives

- If the content of the memory location specified by load linked is changed before the store conditional to the same location is effected, store conditional fails;
- If between the two instructions the processor executes a Context Switch, store conditional fails;
- Store conditional: returns a value indicating whether it succeeded or not (1 if it succeeded, 0 otherwise); load linked returns the initial value.
From Synch. Primitives to Synch. Mechanisms
Spinlock Synchronization Mechanisms

- Starting from atomic operations, coherence mechanisms are used to implement spin locks
  - locks which one processor continues trying to acquire, “spinning” in a loop until it succeeds;
- Spin lock: used when the programmer expects that the lock will be retained for a very short time and wants the locking process (when lock is available) to be a low-latency one.
Spin Lock Synchronization

Read lock variable

Unlocked? (=0?)

Try to lock variable using swap: read lock variable and set it to locked value (1)

Succeed? (=0?)

Unlock variable: set lock variable to 0

Finish update of shared data

atomic operation

Begin update of shared data
Example Test-and-Set (TAS) Spinlock: TAS Function--

```java
public class TASRegister extends Register {
    int value;

    public synchronized int TAS() {
        int result = value;
        value = 1;
        return result;
    }
}
```
Example Test-and-Set (TAS) Spinlock: TAS Function

```java
public class TASRegister extends Register {
    int value;

    public synchronized int TAS() {
        int result = value;
        value = 1;
        return result;
    }
}
```

*remember old value*
Example Test-and-Set (TAS) Spinlock: TAS Function

```java
public class TASRegister extends Register {
    int value;

    public synchronized int TAS() {
        int result = value;
        value = 1;
        return result;
    }
}
```

*set value is 1*
Example Test-and-Set (TAS) Spinlock: TAS Function

```java
public class TASRegister extends Register {
    int value;

    public synchronized int TAS() {
        int result = value;
        value = 1;
        return result;
    }
}
```

return old value
Example Test-and-Set (TAS) Spinlock:
Implementing the locking mechanism

```java
public class TASLOCK implements Lock {
    TASRegister lock = TASRegister(0);

    public void acquire(int i) {
        while (lock.TAS() == 1) {};
    }

    public void release(int i) {
        lock.write(0);
    }
}
```

Keep trying until lock acquired
Example Test-and-Set (TAS) Spinlock: Implementing the locking mechanism

```java
public class TASLOCK implements Lock {
    TASRegister lock = TASRegister(0);

    public void acquire(int i) {
        while (lock.TAS() == 1) {
        }
    }

    public void release(int i) {
        lock.write(0);
    }
}
```

Simple write to release
Performance of Synchronization Mechanisms

- Performances considered in the case of locks:
  - **Low latency**: if a lock is free and no other processor tries to acquire it, requesting processor must acquire it in a short time;
  - **Low traffic**: if many processors simultaneously try acquiring a lock, they must obtain it in sequence generating as little bus traffic as possible;
  - **Scalability**
  - **Low memory cost**
  - "Fairness": ideally, all processors should be able to acquire the lock in the same order in which they requested it.
Synchronization Mechanisms

- E.g.: performances of a lock based on atomic exchange:
  - Very low latency if the same processor acquires the lock multiple times in succession, as instructions are few and lock stays in the processor’s cache;
  - In the case of competition, there is much bus traffic;
  - Limited scalability with increasing numbers of competing processors;
  - Low memory occupation.
Spinlock Synchronization Mechanisms

- The simplest “spin lock” solution is not well-suited for extension to large multiprocessors - if all processors try accessing the same lock, the directory or the bus (depending on the case) act as the serialization mechanism and there ensues an excessive traffic, besides contention problems. Time required for each lock-unlock pair grows exponentially.
Performance Degradation due to locking: Ideal vs Real Locking

Initial speedup: loop overhead in parallel
Performance Degradation due to locking: Ideal vs Real Locking
Further synchronization mechanism (used in particular for programs with parallel loops): barrier; a global mechanism, relating to a number $p$ of processes;

SW. algorithms implementing barriers: use locks, shared counters and flags.

We will examine the centralized barrier.
Simple Video Game

- Prepare frame for display
  - By graphics coprocessor
- “soft real-time” application
  - Need at least 35 frames/second
  - OK to mess up rarely
while (true) {
    frame.prepare();
    frame.display();
}

- What about overlapping work?
  - 1\textsuperscript{st} thread displays frame
  - 2\textsuperscript{nd} prepares next frame
while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}

while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}
Two-Phase Rendering

while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}

while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}

Even phases
Two-Phase Rendering

```java
while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}
```

odd phases

```java
while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}
```
Ideal Parallel Computation
Real-Life Parallel Computation

Cristina Silvano, Sotirios Xydis - Politecnico di Milano - May 2013
Uh, oh
Barrier Synchronization - Desired Behavior
Barrier Synchronization - Desired Behavior
Barrier Synchronization - Desired Behavior

Until every thread has left here

No thread enters here
Centralized Barrier Synchronization

- A centralized counter holds the number of processes that arrived at the barrier: for each arriving process, counter is incremented. Increments must be mutually exclusive;
- After each increment, a process checks whether number $p$ of processes has been reached;
  - No: keeps waiting (busy-wait) on a flag associated with the barrier;
  - Yes: sets the flag to release the $p-1$ waiting processes.
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {

    AtomicInteger count;
    int size;

    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }

    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

Centralized Barrier based on shared counter

Number threads not yet arrived
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

Centralized Barrier based on shared counter
Number threads participating
public class Barrier {
  AtomicInteger count;
  int size;
  public Barrier(int n) {
    count = AtomicInteger(n);
    size = n;
  }
  public void await() {
    if (count.getAndDecrement() == 1) {
      count.set(size);
    } else {
      while (count.get() != 0);
    }
  }
}

Centralized Barrier based on shared counter

Initialization
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

Centralized Barrier based on shared counter

Principal method
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

If I’m last, reset fields for next time
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
What’s wrong with this protocol?

➔ Problem if we want to re-use the barrier (use the same barrier more than once), i.e. one thread “wraps around” to start phase 2 While another thread is still waiting for phase 1

➔ This protocol works only if a new barrier is instantiated for each synchronization point, causing waste of space, code book-keeping
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count; int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

Why this protocol is not working if the barrier object is reused

Phase 1 is so over
Waiting for Phase 1 to finish
Why this protocol is not working if the barrier object is reused

```java
public class Barrier {
    AtomicInteger count;
    int size;

    public Barrier(int n) {
        count = new AtomicInteger(n);
        size = n;
    }

    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
```

Prepare for phase 2
Why this protocol is not working if the barrier object is reused

public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = new AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

Waiting for Phase 1 to finish
Waiting for Phase 2 to finish
Sense-Reversing Barriers

- More elegant and practical solution to the problem of reusing barriers.
- Each SenseBarrier object has a Boolean sense field indicating the sense of the currently executing phase.
- Each thread keeps its current sense as a thread-local object.
- When a thread calls `await()`, it checks whether it is the last thread to decrement the counter. If so, it reverses the barrier’s sense and continues. Otherwise, it spins waiting for the balancer’s sense field to change to match its own local sense.
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {} 
        }
        threadSense.set(!mySense)}}}
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;

    threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}}}
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}}

Store sense for next phase
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)
    }
}

Get new sense determined by last phase
public class Barrier {
  AtomicInteger count;
  int size;
  boolean sense = false;
  threadSense = new ThreadLocal<boolean>...

  public void await {
    boolean mySense = threadSense.get();
    if (count.getAndDecrement()==1) {
      count.set(size); sense = !mySense
    } else {
      while (sense != mySense) {}  
    }
    threadSense.set(!mySense)}}

If I’m last, reverse sense for next time
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {} 
        }
        threadSense.set(!mySense)}}

Otherwise, wait for sense to flip
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}

Prepare sense for next phase
Acknowledgements

- This lecture notes are based on material collected by:
  - Lectures notes on multi-processors from Prof. Sami, PoliMi
  - Companion slides from “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit