Concurrent programs are harder to get right

- Folklore: Need at least an order of magnitude in speedup for concurrent prog to be worth the effort

Some problems are inherently sequential

- Theory – circuit evaluation is P-complete
- Practice – many problems need coordination and communication among sub-problems

Specific issues

- Communication – send or receive information
- Synchronization – wait for another process to act
- Atomicity – do not stop in the middle and leave a mess
Basic question for this course

• How can programming languages make concurrent programming easier?
• Which abstractions are most effective?
  – What are the advantages and disadvantages of various approaches?

This lecture covers past ideas and current Java. Next week we look at forward-looking research ideas.

Basic issue: race conditions

• Sample action
  
  ```
  procedure sign_up(person)
  begin
  number := number + 1;
  list[number] := person;
  end;
  ```

• Problem with parallel execution
  
  ```
  sign_up(fred) || sign_up(bill);
  ```

Resolving conflict between processes

• Critical section
  – Two processes may access shared resource
  – Inconsistent behavior if two actions are interleaved
  – Allow only one process in critical section

• Deadlock
  – Process may hold some locks while awaiting others
  – Deadlock occurs when no process can proceed

Locks and Waiting

<initialize concurrency control>

Thread 1:

  ```
  <wait>
  sign_up(fred); // critical section
  <signal>
  ```

Thread 2:

  ```
  <wait>
  sign_up(bill); // critical section
  <signal>
  ```

Need atomic operations to implement wait

Mutual exclusion primitives

• Atomic test-and-set
  – Instruction atomically reads and writes some location
  – Common hardware instruction
  – Combine with busy-waiting loop to implement mutex

• Semaphore
  – Avoid busy-waiting loop
  – Keep queue of waiting processes
  – Scheduler has access to semaphore; process sleeps
  – Disable interrupts during semaphore operations
    • OK since operations are short

State of the art

• Concurrent programming is difficult
  – Race conditions, deadlock are pervasive

• Languages should be able to help
  – Capture useful paradigms, patterns, abstractions

• Other tools are needed
  – Testing is difficult for multi-threaded programs
  – Many race-condition detectors being built today
    • Static detection: conservative, may be too restrictive
    • Run-time detection: may be more practical for now
**Summary**

- Basic issues in concurrency
  - Race conditions, locking, deadlock, semaphores
- Simple language solutions
  - Cobegin/Coend, Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

**Concurrent language examples**

- Language Examples
  - Cobegin/coend
  - Multilisp futures (skip this year)
  - Actors
  - Concurrent ML (skip this year)
  - Java
- Some features to compare
  - Thread creation
  - Communication
  - Concurrency control (synchronization and locking)

**Cobegin/coend**

- Limited concurrency primitive
- Example
  ```
  x := 0;
  cobegin
  begin x := 1; x := x+1 end;
  begin x := 2; x := x+1 end;
  coend;
  print(x);
  ```
  ![Diagram of concurrent execution]
  Atomicity at level of assignment statement

**Properties of cobegin/coend**

- Advantages
  - Create concurrent processes
  - Communication: shared variables
- Limitations
  - Mutual exclusion: none
  - Atomicity: none
  - Number of processes is fixed by program structure
  - Cannot abort processes
    - All must complete before parent process can go on
  - History: Concurrent Pascal, P. Brinch Hansen, Caltech, 1970's

**Actors**

- Each actor (object) has a script
- In response to input, actor may atomically
  - create new actors
  - initiate communication
  - change internal state
- Communication is
  - Buffered, so no message is lost
  - Guaranteed to arrive, but not in sending order
    - Order-preserving communication is harder to implement
    - Programmer can build ordered primitive from unordered
  - Inefficient to have ordered communication when not needed

**Example**

- Insert 2
  ```
  1, 4, 7
  2, 4, 7
  ```
- Get, Min
  ```
  1, 2, 4, 7
  ```
Actor program

- Stack node parameters
  - a stack_node with acquaintances content and link
  - if operation requested is a pop and content != nil then
    - become forwarder to link
    - send content to customer
  - if operation requested is push(new_content) then
    - let P = new stack_node with current acquaintances
    - become stack_node with acquaintances new_content and P

Hard to read but it does the "obvious" thing, except that the concept of forwarder is unusual....

Forwarder

- Stack before pop
  - 3 — 4 — 5 nil

- Stack after pop
  - forwarder — 4 — 5 nil

- Node "disappears" by becoming a forwarder node. The system manages forwarded nodes in a way that makes them invisible to the program. (Exact mechanism doesn't matter ....)

Concurrency

- Several actors may operate concurrently

- Concurrency not controlled explicitly by program
  - Messages sent by one actor can be received and processed by others sequentially or concurrently

Concurrent ML [Reppy, Gansner, ...]

- Threads
  - New type of entity
- Communication
  - Synchronous channels
- Synchronization
  - Channels
  - Events
- Atomicity
  - No specific language support

Summary

- Basic issues in concurrency
  - Race conditions, locking, deadlock, semaphores
- Simple language solutions
  - Cobegin/Coend, Actor model

Java Concurrency

- Threads
  - Create process by creating thread object
- Communication
  - Shared variables
  - Method calls
- Mutual exclusion and synchronization
  - Every object has a lock (inherited from class Object)
    - synchronized methods and blocks
  - Synchronization operations (inherited from class Object)
    - wait: pause current thread until another thread calls notify
    - notify: wake up waiting threads
Java Thread States

Interaction between threads

- Shared variables
  - Two threads may assign/read the same variable
  - Programmer responsibility
    - Avoid race conditions by explicit synchronization!!
- Method calls
  - Two threads may call methods on the same object
- Synchronization primitives
  - Each object has internal lock, inherited from Object
  - Synchronization primitives based on object locking

Synchronization

- Provides mutual exclusion
  - Two threads may have access to some object
  - If one calls a synchronized method, this locks object
  - If the other calls a synchronized method on same object, this thread blocks until object is unlocked

Synchronized methods

- Marked by keyword
  public synchronized void commitTransaction(...) {}
- Provides mutual exclusion
  - At most one synchronized method can be active
  - Unsynchronized methods can still be called
  - Programmer must be careful
- Not part of method signature
  - sync method equivalent to unsync method with body consisting of a synchronized block
  - Subclass may replace a synchronized method with unsynchronized method

Example

```java
class LinkedCell { // Lisp-style cons cell containing protected double value; // value and link to next cell protected final LinkedCell next;
   public LinkedCell (double v, LinkedCell t) {
      value = v; next = t;
   }
   public synchronized double getValue() { return value; }
   public synchronized void setValue(double v) { value = v; // assignment not atomic }
   public synchronized void setNext(LinkedCell next) { // no synch needed return next; }
}
```
Producer-Consumer?

- Method call is synchronous
- How do we do this in Java?

Solution to producer-consumer

- Cannot be solved with locks alone
  - Use wait and notify methods of Object
- Basic idea
  - Consumer must wait until something is in the buffer
  - Producer must notify waiting consumers when item available
- More details
  - Consumer waits
    • While waiting, must sleep — accomplished with the wait method
    • Need condition recheck loop
  - Producer notifies
    • Must wake up at least one consumer
    • This is accomplished with the notify method

Stack<T>: produce, consume methods

```java
public synchronized void produce(T object) {
  stack.add(object); // Why is loop needed here?
}

public synchronized T consume() {
  while (stack.isEmpty()) {
    try {
      wait(); // Catch InterruptedException e {} {}
    }
    int lastElement = stack.size() - 1;
    T object = stack.get(lastElement);
    stack.remove(lastElement);
    return object;
  }
}
```

See: http://www1.coe.neu.edu/~jsmith/tutorial.html

Limitations of Java 1.4 primitives

- No way to back off from an attempt to acquire a lock
  - Cannot give up after waiting for a specified period of time
  - Cannot cancel a lock attempt after an interrupt
- No way to alter the semantics of a lock
  - Reentrancy, read versus write protection, fairness, ...
- No access control for synchronization
  - Any method can perform synchronized(obj) for any object
- Synchronization is done within methods and blocks
  - Limited to block-structured locking
  - Cannot acquire a lock in one method and release it in another


Concurrency references

- Basic issues in concurrency
  - Race conditions, locking, deadlock, semaphores
- Simple language solutions
  - Cobegin/Coend, Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

More detail in references than required by course
Thread safety

- **Concept**
  - The fields of an object or class always maintain a valid state, as observed by other objects and classes, even when used concurrently by multiple threads.

- **Why is this important?**
  - Classes designed so each method preserves state invariants
  - Example: priority queues represented as sorted lists.
  - Invariants hold on method entry and exit.
  - If invariants fail in the middle of execution of a method, then concurrent execution of another method call will observe an inconsistent state (state where the invariant fails).
  - What’s a “valid state”? Serializability …

---

Example (continued)

```java
public void setColor(int r, int g, int b) {
    checkRGBVals(r, g, b);
    this.r = r;    this.g = g;    this.b = b;
}

public int[] getColor() {
    // returns array of three ints: R, G, and B
    int[] retval = new int[3];
    retval[0] = r; retval[1] = g; retval[2] = b;
    return retval;
}

public void invert() {
    r = 255 - r;    g = 255 - g;    b = 255 - b;
}
```

Question: what goes wrong with multi-threaded use of this class?

---

Example (two slides)

```java
public class RGBColor {
    private int r;    private int g;    private int b;
    public RGBColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r;    this.g = g;    this.b = b;
    }

    private static void checkRGBVals(int r, int g, int b) {
        if (r < 0 || r > 255) g < 0 || g > 255 ||
        b < 0 || b > 255)
            throw new IllegalArgumentException();
    }
}
```

---

Some issues with RGB class

- **Write/write conflicts**
  - If two threads try to write different colors, result may be a “mix” of R,G,B from two different colors.

- **Read/write conflicts**
  - If one thread reads while another writes, the color that is read may not match the color before or after.

---

How to make classes thread-safe

- **Synchronize critical sections**
  - Make fields private.
  - Synchronize sections that should not run concurrently.

- **Make objects immutable**
  - State cannot be changed after object is created.

- **Use a thread-safe wrapper**
  - Use pure functional programming for concurrency.

- **Make thread-safe wrapper**
  - See next slide …

---

Thread-safe wrapper

- **Idea**
  - New thread-safe class has objects of original class as fields.
  - Wrapper class provides methods to access original class object.

- **Example**

```java
public synchronized void setColor(int r, int g, int b) {
    color.setColor(r, g, b);
}

public synchronized int[] getColor() {
    return color.getColor();
}

public synchronized void invert() {
    color.invert();
}
```
### Comparison

- **Synchronizing critical sections**
  - Good default approach for building thread-safe classes
  - Only way to allow `wait()` and `notify()`
- **Using immutable objects**
  - Good if objects are small, simple abstract data type
  - Benefits: pass to methods without alias issues, unexpected side effects
  - Examples: Java String and primitive type wrappers `Integer`, `Long`, `Float`, etc.
- **Using wrapper objects**
  - Can give clients choice between thread-safe version and non-safe
  - Works with existing class that is not thread-safe
  - Example: Java 1.2 collections library – classes are not thread-safe but some have class method to enclose objects in thread-safe wrapper

### Performance issues

- **Why not just synchronize everything?**
  - Performance costs
  - Possible risks of deadlock from too much locking
- **Performance in current Sun JVM**
  - Synchronized method are 4 to 6 times slower than non-synchronized
- **Performance in general**
  - Unnecessary blocking and unblocking of threads can reduce concurrency
  - Immutable objects can be short-lived, increase garbage collector

### Nested monitor lockout problem

- **Background: wait and locking**
  - `wait` and `notify` used within synchronized code
    - Purpose: make sure that no other thread has called method of same object
    - `wait` within synchronized code causes the thread to give up its lock and sleep until notified
    - Allow another thread to obtain lock and continue processing
- **Problem**
  - Calling a blocking method within a synchronized method can lead to deadlock

### Nested Monitor Lockout Example

```java
class Stack {
    LinkedList list = new LinkedList();
    public synchronized void push(Object x) {
        synchronized(list) {
            list.addLast(x); notify();
        }
    }
    public synchronized Object pop() {
        synchronized(list) {
            if(list.size() <= 0) wait();
            return list.removeLast();
        }
    }
}
```

Releases lock on Stack object but not lock on list; a push from another thread will deadlock

### Preventing nested monitor deadlock

- **Two programming suggestions**
  - No blocking calls in synchronized methods, or
  - Provide some nonsynchronized method of the blocking object
- **No simple solution that works for all programming situations**

### Summary

- **Basic issues in concurrency**
  - Race conditions, locking, deadlock, semaphores
- **Simple language solutions**
  - Cobegin/Coend, Actor model
- **Java Concurrency**
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
    - Java memory model
    - Concurrent hash map example
Java Memory Model

- Semantics of multithreaded access to shared memory
  - Competitive threads access shared data
  - Can lead to data corruption
  - Need semantics for incorrectly synchronized programs
- Determines
  - Which program transformations are allowed
    - Should not be too restrictive
  - Which program outputs may occur on correct implementation
    - Should not be too generous

Reference:

Memory Hierarchy

Old memory model placed complex constraints on read, load, store, etc.

Program and locking order

![Program and locking order diagram](Manson, Pugh)

Race conditions

- “Happens-before” order
  - Transitive closure of program order and synchronizes with order
- Conflict
  - An access is a read or a write
  - Two accesses conflict if at least one is a write
- Race condition
  - Two accesses form a data race if they are from different threads, they conflict, and they are not ordered by happens-before

Two possible cases: program order as written, or as compiled and optimized

Race condition question

- “Happens-before” order
  - Transitive closure of program order and synchronizes with order
- Conflict
  - An access is a read or a write
  - Two accesses conflict if at least one is a write
- Race condition
  - Two accesses form a data race if they are from different threads, they conflict, and they are not ordered by happens-before

Subtle issue: program order as written, or as compiled and optimized?

Memory Model Question

- How should the compiler and run-time system be allowed to schedule instructions?
- Possible partial answer
  - If instruction A occurs in Thread 1 before release of lock, and B occurs in Thread 2 after acquire of same lock, then A must be scheduled before B
- Does this solve the problem?
  - Too restrictive: if we prevent reordering in Thread 1,2
  - Too permissive: if arbitrary reordering in threads
  - Compromise: allow local thread reordering that would be OK for sequential programs
Instruction order and serializability

• Compilers can reorder instructions
  – If two instructions are independent, do in any order
  – Take advantage of registers, etc.
• Correctness for sequential programs
  – Observable behavior should be same as if program instructions were executed in the order written
• Sequential consistency for concurrent programs
  – If program P has no data races, then memory model should guarantee sequential consistency
  – Question: what about programs with races?
    • Much of complexity of memory model is for reasonable behavior for programs with races (need to test, debug, ...)

Example program with data race

Sequential reordering + data race

Allowed sequential reordering

Something to prevent ...

Summary of memory model
Volatile fields

- If two accesses to a field conflict:
  - use synchronization to prevent race, or
  - make the field volatile
- gives essential JVM machine guarantees

Consequences of volatile
- reads and writes go directly to memory (not registers)
- volatile longs and doubles are atomic
  - not true for non-volatile longs and doubles
- volatile reads/writes cannot be reordered
  - reads/writes become acquire/release pairs

Volatile happens-before edges

- A volatile write happens-before all following reads of the same variable
  - A volatile write is similar to a lock or monitor exit
  - A volatile read is similar to a lock or monitor enter

Volatile guarantees visibility
- Volatile write is visible to happens-after reads

Volatile guarantees ordering
- Happens-before also constrains scheduling of other thread actions

Other Happens-Before orderings

- Starting a thread happens-before the run method of the thread
- The termination of a thread happens-before a join with the terminated thread
- Many util.concurrent methods set up happen-before orderings
  - placing an object into any concurrent collection happen-before the access or removal of that element from the collection

Example: Concurrent Hash Map

- Implements a hash table
  - Insert and retrieve data elements by key
  - Two items in same bucket placed in linked list
  - Allow read/write with minimal locking
- Tricky
  “ConcurrentHashMap is both a very useful class for many concurrent applications and a fine example of a class that understands and exploits the subtle details of the Java Memory Model (JMM) to achieve higher performance. ... Use it, learn from it, enjoy it — but unless you’re an expert on Java concurrency, you probably shouldn’t try this on your own.”


Summary

- Basic issues in concurrency
  - Race conditions, locking, deadlock, semaphores
- Simple language solutions
  - Cobegin/Coend, Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
- Concurrent hash map example

ConcurrentHashMap

- Concurrent operations
  - read: no problem
  - read/write: OK if different lists
  - read/write to same list: clever tricks sometimes avoid locking
ConcurrentHashMap Tricks

- **Immutability**
  - List cells are immutable, except for data field
  - ⇒ read thread sees linked list, even if write in progress
- **Add to list**
  - Can cons to head of list, like Lisp lists
- **Remove from list**
  - Set data field to null, rebuild list to skip this cell
  - Unreachable cells eventually garbage collected

More info: see homework

Summary

- **Basic issues in concurrency**
  - Race conditions, locking, deadlock, semaphores
- **Simple language solutions**
  - Cobegin/Coend, Actor model
- **Java Concurrency**
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

Basic question for this course

- How can programming languages make concurrent programming easier?
- Which abstractions are most effective?
  - What are the advantages and disadvantages of various approaches?

This lecture covers past ideas and current Java. Next week we look at forward-looking research ideas.