Concurrency
Concurrency

Multiple computations running at the same time

• Concurrency is everywhere, whether we like it or not

• Concurrency is useful, too
  • Splitting up a computation into concurrent pieces is often faster
  • Many apps must handle multiple simultaneous users (e.g. web sites)
  • Even single-user applications are better with concurrency (e.g. Eclipse compiling your Java code in the background while you’re editing it)
Models for Concurrent Programming

Shared Memory

- Analogy: two processors in a computer, sharing the same physical memory

Concurrent modules A and B interact by reading & writing shared state in memory

Message Passing

- Analogy: two computers in a network, communicating by network connections

A and B interact by sending messages to each other through a communication channel
Shared Memory Example

Four customers using cash machines simultaneously

- Shared memory model – each cash machine reads and writes the account balance directly.

Cash machines

A
- deposit $100 to account 1

B
- withdraw $100 from account 2

C
- deposit $100 to account 1

D
- get balance of account 1

Bank

account 1
- $50

account 2
- $200

account 3
- $50

Shared memory
Threads

- A **thread** is a locus of control (i.e. program counter + stack, representing a position in a running program)
  - Simulates a **fresh processor** running the same program in a different place
- A process always has at least one thread (the **main thread**)
- Threads can share any memory in the process, as long as they can get a reference to it
- Threads must set up message passing explicitly (e.g. by creating queues)
Time Slicing

How can I have many concurrent threads with only one or two processors in my computer?

• When there are more threads than processors, concurrency is simulated by **time slicing** (processor switches between threads)
• Time slicing happens unpredictably and nondeterministically

A thread may be paused and resumed at any time
Threads in Java

A thread is represented by java.lang.Thread object

- To define a thread, either override Thread or implement Runnable
  
  T1 extends Thread      R1 implements Runnable

Thread lifecycle

- Starting arguments can be given to the constructor
  
  new T1(arg1, ...)      new Thread(new R1(arg1, ...))

- Thread is spawned by calling its start() method

- New thread starts its life by calling its own run() method

- Thread dies when run() returns or throws an uncaught exception
Race Condition

Suppose A and C run at the same time

A  get balance $50
add deposit  + $100
write back total $150

C  get balance $50
add deposit  + $100
write back total $150

• Neither answer is right!

This is an example of a race condition

• A race condition means that the correctness of the program depends on the relative timing of events in concurrent computations

  • “A is in a race with C”

  • Some interleavings of events may be OK, e.g.: but other interleavings produce wrong answers

Correctness of a concurrent program should not depend on accidents of timing

• Race conditions are nasty bugs -- may be rarely observed, hard to reproduce, hard to debug, but may have very serious effects
Synchronization

**A and C need to synchronize with each other**

- **Locks** are a common synchronization mechanism
- Holding a lock means “I’m changing this; don’t touch it right now”
- Suppose C acquires the lock first; then A must wait to read and write the balance until C finishes and releases the lock
- Ensures that A and C are synchronized, but B can run independently on a different account (with a different lock)

---

**Cash machines**

- **A**
- **B**
- **C**
- **D**

**Bank**

- **Account 1**: $50
- **Account 2**: $200
- **Account 3**: $50 (free)

Shared memory

<table>
<thead>
<tr>
<th>Cash machines</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting for lock</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock holder</td>
<td>C</td>
<td>B</td>
<td>(free)</td>
<td></td>
</tr>
<tr>
<td>Bank</td>
<td>$50</td>
<td>$200</td>
<td>$50</td>
<td></td>
</tr>
<tr>
<td>Account 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Account 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Account 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shared memory
Account Class (no locks)

public class Account {
    long balance;
    void deposit(long amount) { balance += amount; }
    long withdraw(long amount) {
        if (amount <= balance) {
            balance -= amount; return amount;
        } else { long t = balance; balance = 0; return t; }
    }
}

A Closer Look at Deposit

public class Account {
    long balance;
    void deposit(long amount) {
        long t1, t2, t3;
        t1 = amount;
        t2 = balance;
        t3 = t1 + t2;
        balance = t3;
    }
    long withdraw(long amount) { … }
}

© Robert Miller 2008
public interface Lock {
    void lock();
    void unlock();
    boolean tryLock();
    Condition newCondition();
    boolean tryLock(long time, TimeUnit unit);
}
public class Account {
    long balance;
    Lock l = new ReentrantLock();
    void deposit(long amount) {
        l.lock();
        try {
            balance += amount;
        } finally {
            l.unlock();
        }
    }
    long withdraw(long amount) { … }
}
Account Class (with locks)

public class Account {
    long balance; Lock l = new ReentrantLock();
    void deposit(long amount) { … }  
    long withdraw(long amount) {
        l.lock();
        try {
            if (amount <= balance) {
                balance -= amount; return amount;
            } else { long t = balance; balance = 0; return t; }
        } finally { l.unlock(); }
    }
}
Bounded Buffer Example

Producers

A
B
C

Buffer

Consumers

X
Y
Bounded Buffer Example

Producers

A
B
C

Buffer

Consumers

X
Y

A produces 3
Bounded Buffer Example

Producers

A

B

C

Consumers

Buffer

3

X

Y

3 goes into buffer
Bounded Buffer Example

Producers

A
B
C

Consumers

Buffer
X
Y

Y consumes 3
Bounded Buffer Example

Producers
- A produces 4
- B produces 5

Consumers
- X
- Y

Buffer

A produces 4
B produces 5
Bounded Buffer Example

Producers

A

B

C

Consumers

X

Y

4, 5 into buffer (in some order)
Bounded Buffer Example

Producers
- A
- B
- C

Consumers
- X
- Y

Buffer
4
5

C produces 6
C must wait!

In this way producer/consumer rates matched
Bounded Buffer Example

Producers

Consumers

X consumes 5
6 into buffer
C can now resume!
class BoundedBuffer {
    final Object[] items = new Object[2];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        items[putptr] = x;
        if (++putptr == items.length) putptr = 0;
        ++count;
    }

    public Object take() throws InterruptedException {
        Object x = items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;
        return x;
    }
}
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Object[] items = new Object[2];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            items[putptr] = x; if (++putptr == items.length) putptr = 0;
            ++count;
        } finally { lock.unlock(); }
    }

    public Object take() throws InterruptedException { … }
Bounded Buffer (with locks)

class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Object[] items = new Object[2];
    int putptr, takeptr, count;
    public void put(Object x) throws InterruptedException {
        ... }
    public Object take() throws InterruptedException {
        lock.lock();
        try {
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            return x;
        } finally { lock.unlock(); }
    }
}
Condition Variables

Figure out reasons why threads may need to wait
Formulate each reason formally
Each reason is a predicate on state of system

• Bounded buffer is full – producers need to wait
  \((\text{count} == \text{items.length})\)

• Bounded buffer is empty – consumers need to wait
  \((\text{count} == 0)\)

Condition variable for (negation of) each condition

```java
final Condition notFull = lock.newCondition();
final Condition notEmpty = lock.newCondition();
```
Condition Interface

```java
interface Condition {
    void await();
    void signal();
    void signalAll();
}
```

Each condition variable associated with lock

- Condition c = l.newCondition();

Thread must hold lock to call await()

- await() releases lock
- waits until another thread calls signal() or signalAll()
- reaquires lock and returns
Using Condition Variables

Whenever condition may be true, wait until not true

• while (Condition) notCondition.await();
• while (count == items.length) notFull.await(); // inside put
• while (count == 0) notEmpty.await(); // inside take

Whenever condition may become false, signal

public void put(Object x) throws InterruptedException {

...  
items[putptr] = x;
if (++putptr == items.length) putptr = 0;
++count;
notEmpty.signal();

...  
}
Using Condition Variables

Whenever condition may be true, wait until not true

- while (Condition) notCondition.await();
- while (count == items.length) notFull.await(); // inside put
- while (count == 0) notEmpty.await(); // inside take

Whenever condition may become false, signal

public void take(Object x) throws InterruptedException {
  ...

  Object x = items[takeptr];
  if (++takeptr == items.length) takeptr = 0;
  --count; return x;
  notFull.signal();

  ...
}
Bounded Buffer (with condition vars)

class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final Object[] items = new Object[2];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length) notFull.await();
            items[putptr] = x; if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally { lock.unlock(); }
    }

    public Object take() throws InterruptedException { … }
}
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final Object[] items = new Object[2]; int putptr, takeptr, count;
    public void put(Object x) throws InterruptedException { … }
    public void take(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == 0) notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally { lock.unlock(); }
    }
}
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final Object[] items = new Object[2];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            if (count == items.length) notFull.await();
            items[putptr] = x; if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally { lock.unlock(); }
    }

    public Object take() throws InterruptedException { … }
}
Bounded Buffer (subtle error)

class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final Object[] items = new Object[2]; int putptr, takeptr, count;
    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            if (count == 0) notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count; return x;
            notFull.signal();
        } finally { lock.unlock(); }
    }
    public void take(Object x) throws InterruptedException {
        lock.lock();
        try {
            if (count == 0) notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count; return x;
            notFull.signal();
        } finally { lock.unlock(); }
    }
}
Condition Variable Usage Pattern

```java
lock.lock();
try {
    while (...) someCondition.await();
    // some operation on state
    someOtherCondition.signal();
} finally { lock.unlock(); }
```

**Always use while loop around await!!**
Deadlocks

Suppose A and B are making simultaneous transfers

- A transfer between accounts needs to lock both accounts, so that money can’t disappear from the system
- A and B each acquire the lock on the “from” account
- Now each must wait for the other to give up the lock on the “to” account
- Stalemate! A and B are frozen, and the accounts are locked up.

“Deadly embrace”

- Deadlock occurs when concurrent modules are stuck waiting for each other to do something
- A deadlock may involve more than two modules (e.g., a cycle of transfers among N accounts)
- You can have deadlock without using locks – example later
Lock Granularity

Preventing the deadlock

• One solution is to change the locking granularity – e.g. use one lock on the entire bank, instead of a lock on each account

<table>
<thead>
<tr>
<th>$50</th>
<th>$200</th>
<th>$50</th>
</tr>
</thead>
</table>

one lock per account

<table>
<thead>
<tr>
<th>$50</th>
<th>$200</th>
<th>$50</th>
</tr>
</thead>
</table>

one lock for the whole bank

Issues in choosing lock granularity

• If locking is too coarse, then you lose concurrency (e.g. only one cash machine can run at a time)
• If locking is too fine, then you get race conditions and/or deadlocks
• Easy to get this wrong
Deadlock Prevention Strategy

Order locks

Acquire locks in order

In example, order may just be same as account number

```java
to.lock(); from.lock();
```

```java
} else { to.lock(); }
```

```java
// execute transfer
```

```java
} else { to.unlock(); from.unlock(); }
```
Lock and Condition Variable

Operation updates multiple pieces of state
But may need to wait during condition

• conditionVariable = lock2.newCondition();

• lock1.lock();
• lock2.lock();
• state update part 1
• while (need to wait) conditionVariable.await();
• state update part 2
• lock1.unlock();
• lock2.unlock();

What can go wrong here?
How do you fix it?
Nested Monitor Problem

Monitors like an abstract data type
But encapsulate synchronization along with state and operations on state
For example, BoundedBuffer is a monitor
Would like to be able to compose monitors
For example, what about a CountingBoundedBuffer that counts total items ever put into buffer?
Counting Bounded Buffer

CountingBoundedBuffer

total 50

BoundedBuffer

Waiting Objects
Counting Bounded Buffer

class CountingBoundedBuffer {
    int total;  Lock l = new ReentrantLock();
    BoundedBuffer b = new BoundedBuffer();
    public void put(Object x) throws InterruptedException {
        l.lock();
        try { total++;  b.put(x);  } finally { l.unlock();  }
    }
    public void take(Object x) throws InterruptedException {
        Object r;
        l.lock();
        try { r = b.take();  return r; } finally { l.unlock();  }
    }
    public int total() throws InterruptedException {
        l.lock();
        try { return total; } finally { l.unlock();  }
    }
}
Nested Monitor Anomaly

CountingBoundedBuffer deadlocks when BoundedBuffer waits!
How to solve this problem?
Nobody knows a good solution...
Big modularity problem...
Locking Design

• Identify shared and private state
• For each piece of shared state, determine if mutable or read-only (after initialization as private state)
• For each piece of mutable state, identify lock that controls
• Identify lock order (to avoid deadlock)
• Identify atomic operations on mutable state
• For each atomic operation, identify locks to acquire
  • Before operation begins, acquire locks in order
  • After operation ends, release locks
• Identify conditions when operations may need to wait
  • Identify condition variable for each condition
  • Acquire lock, while loop around calls to await
  • Signal whenever condition may become false
• Beware nested monitor anomalies!
Message Passing Example

Modules interact by sending messages to each other

- Incoming requests are placed in a **queue** to be handled one at a time
- Sender doesn’t stop working while waiting for an answer to its request; it handles more requests from its own queue
- Reply eventually comes back as another message

Accounts are now modules, not just memory locations

<table>
<thead>
<tr>
<th>Account 1</th>
<th>Account 2</th>
<th>Account 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>bal: $50</td>
<td>bal: $200</td>
<td>bal: $50</td>
</tr>
</tbody>
</table>

- deposit $100 to account 1
- withdraw $100 from account 2
- deposit $100 to account 1
- get balance of account 1
Message Passing Has the Same Risks

Message passing doesn’t eliminate race conditions

• Suppose the account state machine supports `get-balance` and `withdraw` operations (with corresponding messages)

• Can Alice and Bob always stay out of the OVERDRAWN state?

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>get-balance</td>
<td>get-balance</td>
</tr>
<tr>
<td>if balance &gt; $75,</td>
<td>if balance &gt; $50,</td>
</tr>
<tr>
<td>withdraw $75</td>
<td>withdraw $50</td>
</tr>
</tbody>
</table>

- Lesson: need to carefully choose the atomic (indivisible) operations of the state machine – `withdraw-if-sufficient-funds` would be better

Account bal: $100

Message-passing can have deadlocks too

• Particularly when using finite queues that can fill up
Concurrency Is Hard to Test

**Poor coverage**

- Recall our notions of coverage
  - all states, all transitions, or all paths through a state machine
- Given two concurrent state machines (with N states and M states), the combined system has N x M states (and many more transitions and paths)
- As concurrency increases, the state space explodes, and achieving sufficient coverage becomes infeasible

**Poor reproducibility**

- Transitions are *nondeterministic*, depending on relative timing of events that are strongly influenced by the environment
  - Delays can be caused by other running programs, other network traffic, operating system scheduling decisions, variations in processor clock speed, etc.
- Test driver can’t possibly control all these factors
- So even if state coverage were feasible, the test driver can’t reliably reproduce particular paths through the combined state machine
We’ll focus on message passing, not shared memory

• Locking strategy for shared-memory paradigm can be challenging
• Message-passing paradigm often aligns directly with the real-world workflow of a problem
• But message passing is less suited to some problems, e.g. a big shared data structure
Message Passing with Threads

Use a synchronized queue for message-passing between threads

- interface java.util.concurrent.BlockingQueue is such a queue

ArrayBlockingQueue is a fixed-size queue that uses an array representation

LinkedBlockingQueue is a growable queue (no FULL state) using a linked-list representation

no \texttt{take}-transition in EMPT\textsc{y} state, so a thread that tries to \texttt{take} from an empty queue must \textbf{block} (wait) until it can
Message Queues with Tasks

Task is some piece of computation to execute
Often useful to structure computation as

• Set of threads
• Interact via message queues
• Message queues contain tasks

Common pattern

• Shared object
• Message queue in front of object
• All tasks that execute access that object
• Serialize access to object

General formulation

Supports extensibility
Task Interface

interface Task<T> {
    public void execute(T o);
}
import java.util.concurrent.*;

class Worker<T> {
    BlockingQueue<Task<T>> q = new LinkedBlockingQueue<Task<T>>();

    Worker(final T o) {
        new Thread() {
            public void run() {
                while (true) {
                    try { Task<T> t = q.take(); t.execute(o); } catch (Exception e) {} 
                }
            }
        }.start();
    }

    public void put(Task<T> t) {
        try { q.put(t); } catch (Exception e) {} 
    }
}

Using Worker Class

```java
public class Main {
    public static void main(String[] args) {
        final Worker<Counter> w =
            new Worker<Counter>(new Counter());
        new Thread() { public void run() {
            w.put( new Task<Counter>() {
                public void execute(Counter c) { c.increment(); }
            });
        }}.start();
        new Thread() { public void run() {
            w.put( new Task<Counter>() {
                public void execute(Counter c) { c.increment(); }
            });
        }}.start();
    }
}
```
Case Study: Photo Organizer

What happens when the UI displays a large album?
Concurrent in GUIs

Mouse and keyboard events are accumulated in an event queue

• Event loop reads an input event from the queue and dispatches it to listeners on the view hierarchy

• In Java, the event loop runs on a special *event-handling thread*, started automatically when a user interface object is created
Java Swing Is Not Threadsafe

The view hierarchy is a big meatball of shared state

• And there’s no lock protecting it at all
• It’s OK to access user interface objects from the event-handling thread (i.e., in response to input events)
• But the Swing specification forbids touching – reading or writing – any Component objects from a different thread
  • See “Threads and Swing”,
  • The truth is that Swing’s implementation does have one big lock (Component.getTreeLock()) but only some Swing methods use it (e.g. layout)
Message Passing Via the Event Queue

The event queue is also a message-passing queue

- To access or update Swing objects from a different thread, you can put a message (represented as a Runnable object) on the event queue
  ```java
  SwingUtilities.invokeLater(new Runnable() {
      public void run() { content.add(thumbnail); ...} });
  ```
- The event loop handles one of these pseudo-events by calling run()
Thread Safety

BlockingQueue is itself a shared state machine

- But it’s OK to use from multiple threads because it has an internal lock that prevents race conditions within the state machine itself
  - So state transitions are guaranteed to be atomic
  - This is done by the Java synchronized keyword

- BlockingQueue is therefore thread-safe (able to be called by multiple threads safely without threat to its invariants)
- HashSet is not thread-safe; neither is the Swing view hierarchy
Other Thread-Safe Classes

Lists, Sets, and Maps can be made thread-safe by a wrapper function

• \( t = \text{Collections.synchronizedSet}(s) \) returns a thread-safe version of set \( s \), with a lock that prevents more than one thread from entering it at a time, forcing the others to block until the lock is free

• So we could imagine synchronizing all our sets:

```java
thumbnails = Collections.synchronizedSet(new HashSet<Thumbnail>());
```

**This doesn’t fix all race conditions!**

• Doesn’t help preserve invariants involving more than one data structure

```java
thumbnails.add(t);
content.add(t);
```

these operations need to be atomic together, to avoid breaking the rep invariant of PreviewPane (that all thumbnails are children of content)
More Thread-Safe Classes

Objects that never change state are usually* thread-safe

• **Immutable** objects never change state
  • e.g., java.lang.String is immutable, so threads can share strings as much as they like without fear of race conditions, and without any need for locks or queues

* Caveat: some apparently immutable objects may have hidden state: e.g. memoizing (caching) method return values.
Thread-safe or Not?

Which of the following are thread-safe? If not, how could you ensure that they are thread-safe?

• a `findPrimes()` method that remembers all the primes it’s ever found in an `ArrayList`

• a method that times itself, using a static variable to store its start time

• a method that takes a `String` and replaces all the spaces in it with underscores

• a method that takes an integer array and replaces all zeroes in it with ones
Summary

Concurrency
• Multiple computations running simultaneously

Shared-memory & message-passing paradigms
• Shared memory needs a synchronization mechanism, like locks
• Message passing synchronizes on communication channels, like queues

Pitfalls
• Race when correctness of result depends on relative timing of events
• Deadlock when concurrent modules get stuck waiting for each other

Design advice
• Share only immutable objects between threads
• Use blocking queues and SwingUtilities.invokeLater()