Concurrency Practices

Saman Amarasinghe
Spring 2008
void TransferAction (Account fracc) throws Exception {
    out.println("Destination Account ID > ");
    String id = in.readLine(); System.out.println(id);
    Account toacc = bnk.get(id);
    if (toacc == null)
        return;

    out.println("your balance is $" + fracc.getbal());
    out.println("Transfer amount > ");
    int val = Integer.valueOf(in.readLine());
    if(val < 0) {
        out.println("Can’t withdraw from other accounts");
        return;
    }

    int frcurr = fracc.getbal();
    if(frcurr - val < 0) {
        out.println("Insufficient Balance");
        return;
    }

    int tocurr = toacc.getbal();
    frcurr = frcurr - val;
    tocurr = tocurr + val;
    toacc.setbal(tocurr);

    fracc.setbal(frcurr);
    out.println("New balance is $" + frcurr);
}
void TransferAction (Account fracc) throws Exception {
    out.println("Destination Account ID > ");
    String id = in.readLine(); System.out.println(id);
    Account toacc = bnk.get(id);
    if (toacc == null)
        return;

    out.println("your balance is $" + fracc.getbal());
    out.println("Transfer amount > ");
    int val = Integer.valueOf(in.readLine());
    if(val < 0) {
        out.println("Can’t withdraw from other accounts");
        return;
    }

    synchronized(fracc) {
        int frcurr = fracc.getbal();
        if (frcurr - val < 0) {
            out.println("Insufficient Balance");
            return;
        }
    }

    synchronized(toacc) {
        int tocurr = toacc.getbal();
        frcurr = frcurr - val;
        tocurr = tocurr + val;

        toacc.setbal(tocurr);
    }

    fracc.setbal(frcurr);
    out.println("New balance is $" + frcurr);
}

© Saman Amarasinghe 2008
public class Account {
    String id;
    String password;
    int balance;
    private static int currOrd = 0;
    private final int ord;

    Account(String id, String password,
            int balance) {
        this.id = id;
        this.password = password;
        this.balance = balance;
        this.ord = currOrd++;
    }

    boolean is_password(String password) {
        return password.equals(this.password);
    }

    int getbal() {
        return balance;
    }

    void setbal(int v) {
        balance = v;
    }

    int getord() {
        return ord;
    }
}

void TransferAction (Account fracc) throws Exception {
    String id = in.readLine(); System.out.println(id);
    Account toacc = bnk.get(id);
    if (toacc == null)
        return;
    out.println("your balance is "$ + fracc.getbal());
    out.println("Transfer amount > ");
    int val = Integer.valueOf(in.readLine());
    if(val < 0) {
        out.println("Can’t withdraw from other accounts");
        return;
    }
    Account lk1 = (fracc.getord() > toacc.getord())?fracc:toacc;
    Account lk2 = (fracc.getord() > fracc.getord())?toacc:fracc;
    synchronized(lk1) {
        synchronized(lk2) {
            int frcurr = fracc.getbal();
            if (frcurr - val < 0) {
                out.println("Insufficient Balance");
                return;
            }
            int tocurr = toacc.getbal();
            frcurr = frcurr - val;
            tocurr = tocurr + val;
            toacc.setbal(tocurr);
            fracc.setbal(frcurr);
            out.println("New balance is "$ + frcurr);
        }
    }
}

Is this a fair scheme?
Why use Concurrent Programming?

1. Natural Application Structure
   - The world is not sequential! Easier to program multiple independent and concurrent activities.

2. Increased application responsiveness
   - Not blocking the entire application due to blocking IO

3. Performance from multiprocessors and multicores
   - Parallel execution

4. Distributed systems
   - Single application on multiple machines
   - Client/server type or peer-to-peer systems
I. Natural Application Structure

Many problems are easier to decompose as multiple concurrent tasks

- Easier to specify
- Reduces interdependencies
- Easier to implement
- Easier to debug and test

Examples?
2. Increased application responsiveness

**Gracefully handle long delays**
- Blocking IO is simpler than non-blocking IO
- More responsive user interfaces

**Examples?**
3. Performance from Multi-processors/Multicores

### Multicores

**From supercomputers to desktops to laptops – they are everywhere**

- **Paradigm shift on the way**
  - Before: every 18 months single processor performance doubles
  - Now: minimal improvement in single processor performance, number of cores will double every 18 months.

- **In the future: more performance → parallelize**
  - Burden on the programmer
  - Expect to see a lot more parallel programs

### Multiprocessors

**Powerful workstation to Supercomputers**

- Solving very large problems
  - Weather simulation for hurricane prediction
  - Finite element analysis of an aircraft body
  - Protein folding for drug design

- Fastest computer on the planet:
  - IBM BlueGene/L at Lawrence Livermore National Labs
  - 212,992 processors!

---

© Saman Amarasinghe 2008
Issues with Parallelism

**Amdhal’s Law**
- Any computation can be analyzed in terms of a portion that must be executed sequentially, $T_s$, and a portion that can be executed in parallel, $T_p$.
- Then for $n$ processors:
  - $T(n) = T_s + \frac{T_p}{n}$
  - $T(\infty) = T_s$, thus maximum speedup $(T_s + T_p) / T_s$

**Load Balancing**
- The work is distributed among processors so that all processors are kept busy all of the time.

**Granularity**
- The size of the parallel regions between synchronizations or the ratio of computation (useful work) to communication (overhead).
Implications of Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{N}}
\]

\[
\text{Utilization} = \frac{1}{p + N(1 - p)}
\]
4. Distributed Systems

Solving Google scale problems
- web search
- Mail service (gmail, yahoo mail)
- Video distribution (utube)

What is the difference between distributed systems and supercomputers?
How to build a concurrent program?

How to enforce safety of the objects in a concurrent program?

How to use different design patterns to compose concurrent “parts” and build a program?

How to use different schemes for assigning parallel work to threads?
Enforcing Safety of Objects

Object Safety: Having a sharing strategy for each object

- Thread-confined
- Single owner
- Shared read-only
- Shared thread-safe
- Guarded by others

© Saman Amarasinghe 2008
Thread Confined

The object is exclusively held by one thread. No other threads get to see that object.

Implementation Issues

- Don’t let the object “escape” the scope of the thread
- If threads are heterogeneous, use type system to restrict visibility (objects of a class accessed only within one thread)
- Local variables will not escape the function
- Mark the objects with the thread id and use an assertion to check

Pros and Cons?
II. Single Owner

At any given time exactly one thread owns and accesses the object. An explicit handoff of the object between the threads.

Implementations Issues

- Use a thread safe class for handoff
- Minimize number of pointers to the object while in use

Pros and Cons?
III. Shared Read-Only

Immutable objects \(\rightarrow\) thread safe

Functional Programming Style

- Will revisit in later lectures

Implementations Issues

- Final fields
- Only constructors can modify the fields

Pros and Cons?
IV. Shared Thread-Safe

Build a properly synchronized class or Extend or use one of Java’s thread-safe class.

Implementations Issues

- All fields should be either final or protected
- All methods should be synchronized
- Atomic actions should have their own method

Pros and Cons?
V. Guarded by Others

All the methods that access the object has to properly guard it.

Implementations Issues

- Have a lock designated to guard the object
- Use this if synchronization cannot be done within the object

Pros and Cons?
public class MutablePoint {
    public int x, y;
    public MutablePoint() {
        x = 0;
        y = 0;
    }
    public MutablePoint(MutablePoint p) {
        this.x = p.x;
        this.y = p.y;
    }
    public String toString() {
        return "[" + x + ", " + y + "]";
    }
}

public class VehicleTracker {
    private final Map<String, MutablePoint> locations;
    public VehicleTracker(Map<String, MutablePoint> locations) {
        this.locations = new HashMap<String, MutablePoint>();
        for (String id : locations.keySet())
            this.locations.put(id, new MutablePoint(locations.get(id)));
    }
    public Map<String, MutablePoint> getLocations() {
        return locations;
    }
    public MutablePoint getLocation(String id) {
        MutablePoint loc = locations.get(id);
        return loc;
    }
    public void setLocation(String id, int x, int y) {
        MutablePoint loc = locations.get(id);
        if (loc == null)
            throw new IllegalArgumentException("No such ID:" + id);
        loc.x = x;
        loc.y = y;
    }
    public String toString() {
        String res = "";
        for (String id : locations.keySet())
            res = res+
"("+id +","+locations.get(id).toString()+ ") ";
        return res;
    }
}

public class main {
    static private VehicleTracker vt;
    public static void main(String[] args) {
        Map<String, MutablePoint> loc =
            new HashMap<String, MutablePoint>();
        loc.put("Daniel", new MutablePoint());
        loc.put("Saman", new MutablePoint());
        vt = new VehicleTracker(loc);
        play();
    }
    static void play() {
        vt.setLocation("Saman", 10, 100);
        vt.setLocation("Daniel", 5, 5);
        System.out.println(vt.toString());
        Point p = vt.getLocation("Saman");
        p.x = 100;
    }
}

© Saman Amarasinghe 2008
Tracking-seq

DEMO I

© Saman Amarasinghe 2008
public class VehicleTracker {
    private final Map<String, MutablePoint> locations;

    public VehicleTracker(Map<String, MutablePoint> locations) {
        this.locations = new HashMap<String, MutablePoint>();
        for (String id : locations.keySet())
            this.locations.put(id, new MutablePoint(locations.get(id)));
    }

    public Map<String, MutablePoint> getLocations() {
        return locations;
    }

    public MutablePoint getLocation(String id) {
        MutablePoint loc = locations.get(id);
        if (loc == null)
            throw new IllegalArgumentException("No such ID: " + id);
        loc.x = x;
        loc.y = y;
        return loc;
    }

    public String toString() {
        String res = "";
        for (String id : locations.keySet())
            res = res+"("+id+","+locations.get(id).toString()+") " ;
        return res;
    }
}

private static Map<String, MutablePoint> deepCopy(Map<String, MutablePoint> m) {
    Map<String,MutablePoint> result = new
    HashMap<String,MutablePoint>();
    for (String id : m.keySet())
        result.put(id, new MutablePoint(m.get(id)));
    return result;
}
Tracking-no-rep-exposure

DEMO II

© Saman Amarasinghe 2008
Design patterns for concurrent programs

How to compose different parallel objects

Use well studied design patterns

- Monitor pattern
- Delegation pattern
- Divide and conquer pattern (time permitting)
Monitor Pattern

Encapsulate all the mutable state and guards it with the object’s own lock

The object is thread safe

All the methods can be viewed as “atomic” actions

All mutable parts of the object cannot escape
public class VehicleTracker {
    private final Map<String, MutablePoint> locations;

    public VehicleTracker(Map<String, MutablePoint> locations) {
        this.locations = deepCopy(locations);
    }

    public synchronized Map<String, MutablePoint> getLocations() {
        return deepCopy(locations);
    }

    public synchronized MutablePoint getLocation(String id) {
        Point loc = locations.get(id);
        return (loc == null)?null: new MutablePoint(loc);
    }

    public synchronized void setLocation(String id, int x, int y) {
        MutablePoint loc = locations.get(id);
        if (loc == null) throw new IllegalArgumentException("No such ID: " + id);
        loc.x = x;
        loc.y = y;
    }

    public synchronized String toString() {
        String res = "";
        for (String id : locations.keySet())
            res = res + "(" + id + ", " + locations.get(id).toString() + ") " + locations.get(id).toString() + ");"
        return res;
    }

    private static Map<String, MutablePoint> deepCopy(Map<String, MutablePoint> m) {
        Map<String, MutablePoint> result = new HashMap<String, MutablePoint>();
        for (String id : m.keySet())
            result.put(id, new MutablePoint(m.get(id)));
        return result;
    }
}

© Saman Amarasinghe 2008
Tracking-monitor-pattern

DEMO III

© Saman Amarasinghe 2008
Questions

How did we enforce safety of the MutablePoint class?

What properties of the program that were influential in making this concurrent code correct?

What are the Pros and Cons of the Monitor Pattern?
Delegation Pattern

Let someone else do the hard work!
Most objects are stored in Java collections
Thread safe collections are available → use them

- List
- Map
- Set
- SortedMap
- SortedSet
Mutable and Immutable Objects

```java
public class MutablePoint {
    public int x, y;

    public MutablePoint() {
        x = 0;
        y = 0;
    }

    public MutablePoint(Point p) {
        this.x = p.x;
        this.y = p.y;
    }

    public String toString() {
        return "[" + x + ", " + y + "]";
    }
}

public class ImmutablePoint {
    public final int x, y;

    public ImmutablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public ImmutablePoint(ImmutablePoint p) {
        this.x = p.x;
        this.y = p.y;
    }

    public String toString() {
        return "[" + x + ", " + y + "]";
    }
}
```

© Saman Amarasinghe 2008
public class VehicleTracker {
    private final ConcurrentMap<String, ImmutablePoint> locations;
    private final Map<String, ImmutablePoint> unmodifiableMap;

    public VehicleTracker(Map<String, ImmutablePoint> points) {
        locations = new ConcurrentHashMap<String, ImmutablePoint>(points);
        unmodifiableMap = Collections.unmodifiableMap(locations);
    }

    public Map<String, ImmutablePoint> getLocations() {
        return unmodifiableMap;
    }

    public ImmutablePoint getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (locations.replace(id, new ImmutablePoint(x, y)) == null)
            throw new IllegalArgumentException("invalid vehicle name: "+id);
    }
}
Questions

How did we enforce the safety of the ImmutablePoint class?

What are the advantages/disadvantages of using MutablePoint vs. ImmutablePoint?
File Server Example

Can open a connection to the server and give a file name

Server will fetch that file and send the data back over the connection
public class request {
    private Socket soc;
    private String fileName;
    private List<String> data = new ArrayList<String>();

    request(Socket soc) {
        this.soc = soc;
    }

    public void getFileName() {
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(soc.getInputStream()));
            fileName = in.readLine();
        } catch (Exception e) { System.err.println("Socket error"); }
    }

    public void readFile() {
        try {
            FileInputStream fstream = new FileInputStream(fileName);
            DataInputStream in = new DataInputStream(fstream);
            while (in.available() != 0) {
                data.add(in.readLine());
            }
            in.close();
        } catch (Exception e) { System.err.println("File read error"); }
    }

    public void sendData() {
        try {
            PrintWriter out = new PrintWriter(soc.getOutputStream(), true);
            for (String c : data) {
                out.println(c);
            }
            soc.close();
        } catch (Exception e) { System.err.println("Socket error"); }
    }
}
public class fileserver {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(3333);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}

private static void handleRequest(Socket connection) {
    Request req = new Request(connection);
    req.getFileName();
    req.readFile();
    req.sendData();
}
Filer-sequential

DEMO IV
Schemes for Assigning Work to Threads

Once parallel parts of the program are identified, need to assign them to threads.

- Thread per Task
- Using a Thread Pool
- Stream Data
Thread per Task

When we reach a fork in control flow where each branch can be run concurrently

- Start a thread to run one branch
- Run the other branch on the original thread

Exposes the maximum amount of concurrency to run in parallel
public class fileserver {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(3333);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}

private static void handleRequest(Socket connection) {
    request req = new request(connection);
    req.getFileName();
    req.readFile();
    req.sendData();
}

public class fileserver {
    public static void main(String[] args) throws IOException {
        final ServerSocket socket = new ServerSocket(3333);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }
}

private static void handleRequest(Socket connection) {
    request req = new request(connection);
    req.getFileName();
    req.readFile();
    req.sendData();
}

How is safety enforced in the request class?
Using a Thread Pool

Create a fixed number of threads

When we reach a fork in control flow where each branch can be run concurrently

- Run one branch on the original thread
- If a thread is available, run the other branch on that thread
- Otherwise, wait till a thread becomes free
Thread per Task

```java
public class fileserver {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(3333);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}
```

Using a Thread Pool

```java
public class fileserver {
    private static final int NTHREADS = 10;
    private static final Executor exec = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        final ServerSocket socket = new ServerSocket(3333);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
```
Streaming Data

Program can be divided into a graph of segments, where each segment

- Consume the data produced by the previous segment
- Do its processing on the data and produce the output
- The output is sent to the next segment
Streaming Data

Program can be divided into a graph of segments, where each segment

- Consume the data produced by the previous segment
- Do its processing on the data and produce the output
- The output is sent to the next segment

Can be implemented using Queues

Multiple threads can execute each segment

- Thread pool per segment
public class fileserver {
    static private BlockingQueue<request> toGetName, toReadFile, toSendData;
    private static final int NT_GFN = 1;
    private static final int NT_RF = 10;
    private static final int NT_SD = 5;

    public static void main(String[] args) throws IOException {
        toGetName = new LinkedBlockingQueue<request>();
        toReadFile = new LinkedBlockingQueue<request>();
        toSendData = new LinkedBlockingQueue<request>();

        new processGetName(toGetName, toReadFile, NT_GFN);
        new processReadFile(toReadFile, toSendData, NT_RF);
        new processSendData(toSendData, null, NT_SD);

        try {
            ServerSocket socket = new ServerSocket(3333);
            while (true) {
                final Socket connection = socket.accept();
                toGetName.put(new request(connection));
            }
        } catch (Exception e) {  }
    }
}
public abstract class processQueue {
    BlockingQueue<request> inq, outq;

    processQueue() { }

    void start(BlockingQueue<request> inq,
               BlockingQueue<request> outq,
               int numThreads) {
        this.inq = inq;
        this.outq = outq;

        for(int i=0; i<numThreads; i++) {
            Runnable task = new Runnable() {
                public void run() {
                    processAll();
                }
            };
            new Thread(task).start();
        }
    }

    private void processAll() {
        while(true) {
            try {  
                request req = inq.take();
                processOne(req);
                if(outq != null) outq.put(req);
            } catch(Exception e) { }
        }
    }

    abstract void processOne(request req);
}
public abstract class processQueue {
    BlockingQueue<request> inq, outq;

    processQueue() { }

    void start(BlockingQueue<request> inq,
               BlockingQueue<request> outq,
               int numThreads) {
        this.inq = inq;
        this.outq = outq;

        for(int i=0; i<numThreads; i++) {
            Runnable task = new Runnable() {
                public void run() {
                    processAll();
                }
            };
            new Thread(task).start();
        }
    }

    private void processAll() {
        while(true) {
            try {
                request req = inq.take();
                processOne(req);
                if(outq != null) outq.put(req);
            } catch(Exception e) { }
        }
    }

    abstract void processOne(request req);
}

public class processGetName extends processQueue{
    processGetName(BlockingQueue<request> inq,
                    BlockingQueue<request> outq,
                    int nt) {
        start(inq, outq, nt);
    }

    void processOne(request req) {
        req.getName();
    }
}

public class processReadFile extends processQueue{
    processReadFile(BlockingQueue<request> inq,
                    BlockingQueue<request> outq,
                    int nt) {
        start(inq, outq, nt);
    }

    void processOne(request req) {
        req.readFile();
    }
}

public class processSendData extends processQueue{
    processSendData(BlockingQueue<request> inq,
                    BlockingQueue<request> outq,
                    int nt) {
        start(inq, outq, nt);
    }

    void processOne(request req) {
        req.sendData();
    }
}
Filer-stream

DEMO V

© Saman Amarasinghe 2008
Question

How is safety enforced in the request class?

Compare and contrast Thread per Task, Using Thread Pool and Streaming Data
public class FindMaxValue {
    private static final int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    int start, end, result;

    FindMaxValue(int start, int end) {
        this.start = start;
        this.end = end;
        this.result = solveSequentially();
    }

    public static void main(String[] args) {
        FindMaxValue problem = new FindMaxValue(0, numbers.length-1);
        System.out.println("The Maximum Value is "+ problem.result);
    }

    public int solveSequentially() {
        int max = Integer.MIN_VALUE;
        for (int i=start; i<=end; i++) {
            int n = numbers[i];
            if (n > max)
                max = n;
        }
        System.out.println("seq [" + start + ", " + end + "] ->" + max);
        return max;
    }
}
Divide and Conquer Pattern

How to parallelize Recursive Problems

```c
int solve(int low, int high) {
    if(high-low < threshold)
        return solve_sequentially(low, high);
    else {
        int mid = (low+high)/2;
        int lft = solve(low, mid-1);
        int rgt = solve(mid, high);
        return func(lft, rgt);
    }
}
```

Algorithm

- If range is small, just do it.
- Otherwise, run left and right concurrently
- Wait until both are done
- continue
public class FindMaxValue extends Thread {
    private static final int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
    private static final int threshold = 4;  
    int start, end, result;

    FindMaxValue(int start, int end) {
        this.start = start;  this.end = end;
        start();  
    }

    public static void main(String[] args) {
        FindMaxValue problem = new FindMaxValue(0, numbers.length-1);
        try {
            problem.join();  
        } catch(Exception e) { }
    }

    public void run() {
        if(end-start+1 < threshold) {
            result = solveSequentially();  
            return;
        }
        int mid = (start+end)/2;
        FindMaxValue lft = new FindMaxValue(start, mid);
        FindMaxValue rgt = new FindMaxValue(mid+1, end);
        try {
            lft.join();  
            rgt.join();  
            result = Math.max(lft.result, rgt.result);
        } catch(Exception e) { }
    }

    public void solveSequentially() { ........ }
}
Maxval divide-and-conquer

DEMO VI

© Saman Amarasinghe 2008
Questions

How did we enforce safety on the elements of the number array?

If the algorithm had to modify the elements of the number array (example sort), will this scheme makes it unsafe?

Then how is safety enforced?

Do you see any problems with this divide and conquer approach? Is so how would you solve them?
Summary

How to enforce safety of the objects in a concurrent program.
- Thread-confined
- Single owner
- Shared read-only
- Shared thread-safe
- Guarded by others

How to use different patterns to build a concurrent program
- Monitor pattern
- Delegation pattern
- Divide and conquer pattern

How to use different schemes for assigning work to threads
- Thread per Task
- Using a Thread Pool
- Stream Data
Concurrent Programming References

Some examples in this lecture were derived from the book

© Saman Amarasinghe 2008
Lecture Exercises

In the vehicle tracker example, can we use MutablePoint in the Delegation pattern implementation by making a copy or deepcopy before returning the point(s)? Explain.

What pattern is used in the stream data example?

Describe if it makes sense to use each of the safety enforcement technique in a stream data example.

Describe how to use thread pools in implementing a divide and conquer pattern.

Can we use streaming in implementing a divide and conquer pattern?

© Saman Amarasinghe 2008