Object Oriented Programming and Design in Java

Session 23

Instructor: Bert Huang

Announcements

- Homework 5 due last day of class: Mon. May 3rd (in one week)
- Mon. May 3rd: Final review
- Mon. May 10th, Final exam. 9 AM - noon
  - closed-book/notes, focus on post-midterm material, but material is inherently cumulative

Review

- VISITOR pattern
- Networking
- Socket and ServerSocket classes
- Simple text-chat example program

Today’s Plan

- Multithreading with Conditions review (for the homework)
- Multithreading in the chat program
- Sending non-string data over the network
- MVC over the network

Pigeon Threads

- Each pigeon should be controlled by its own thread with infinite loop:
  - find freshest food location
    - block if no food
  - move toward food (with randomness)
  - remove food if touching food
  - randomly get startled

Locks Review

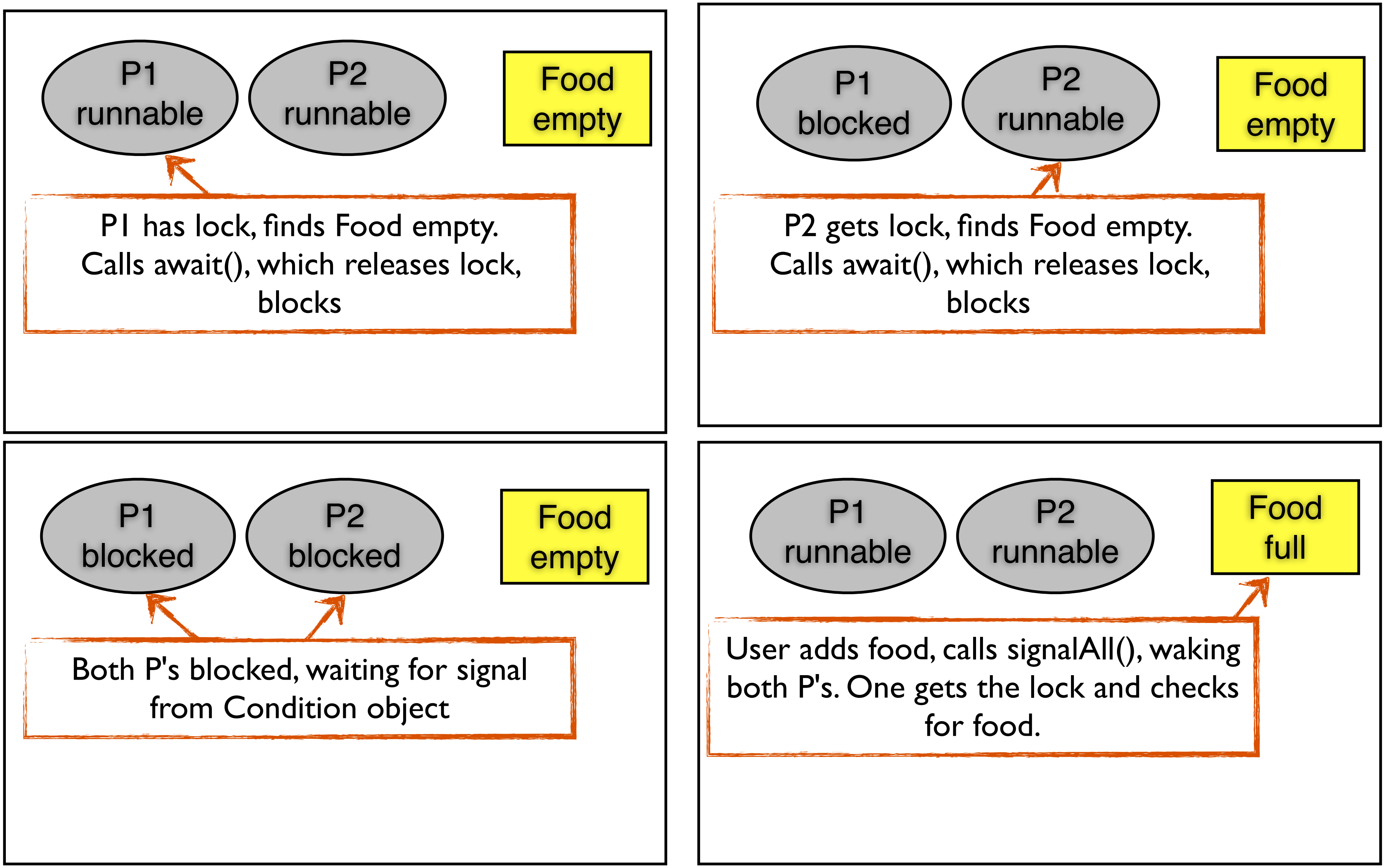
- Each thread must `lock()` a `Lock` object before doing tasks that can cause race conditions
- Once the lock is acquired, the thread may find it cannot operate, e.g., the data structure it wants to remove from is empty
- Then release the lock using a `Condition` object
- After the work is done, `unlock()` the `Lock` inside a `finally { }` block to ensure that it is unlocked even if an exception occurs

Condition Objects

- Each `Lock` can have any number of `Condition` objects
- `Condition setNonEmpty = setLock.newCondition()`

-
  ```
  setLock.lock()
  while(set.isEmpty())
      setNonEmpty.await() // releases the lock
  ```
- Whenever the condition could have changed, call `setNonEmpty.signalAll()`
  - Unblock all waiting threads, but a thread must reacquire the lock before returning from `await`

Pigeons and Food



P1 runnable

P2 runnable

Food empty

P1 has lock, finds Food empty. Calls await(), which releases lock, blocks

P1 blocked

P2 runnable

Food empty

P2 gets lock, finds Food empty. Calls await(), which releases lock, blocks

P1 blocked

P2 blocked

Food empty

Both P's blocked, waiting for signal from Condition object

P1 runnable

P2 runnable

Food full

User adds food, calls signalAll(), waking both P's. One gets the lock and checks for food.

Helpful Links

- http://java.sun.com/docs/books/tutorial/essential/concurrency/newlocks.html
- http://java.sun.com/javase/6/docs/api/java/util/concurrent/locks/Condition.html
- http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html

Multithreading

- These programs work, but the conversation must alternate back and forth between the client and server
- We need multithreading to allow remote messages to be displayed immediately while waiting for System.in input
- ThreadedBufferedReaderPrinter - Runnable: continually prints output from BufferedReader ASAP
- ThreadedChatServer - reads input from console and sends it to client, starts TBRP thread
- ThreadedChatClient - reads input from console and sends it to server, starts TBRP thread

```java
public class ThreadedBufferedReaderPrinter implements Runnable {

   /**
    * Constructor takes the BufferedReader to print
    * @param reader the BufferedReader to print
    */
   public ThreadedBufferedReaderPrinter(BufferedReader reader) {
      this.reader = reader;
   }

   public void run() {
      String line;
      try {
         while (!Thread.interrupted() &&
               (line = reader.readLine()) != null) {
            System.out.println(line);
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
   }

   BufferedReader reader;
}
```

ThreadedBufferedReaderPrinter

ThreadedChatClient Main Loop

```
// hostname and port loaded

TextClient client = new TextClient(hostname, port);

// Start printing thread
Thread t = new Thread(new
     ThreadedBufferedReaderPrinter(client.getReader()));
t.start();

// start chatting
while (client.isConnected()) {
   try {
     client.writeLine(stdin.readLine());
   } catch (IOException e) {
     e.printStackTrace();
   }
}
```

ThreadedChatServer Main Loop

```
// port loaded

TextServer server = new TextServer(port);
server.writeLine("Connected to server");

// Start printing thread
Thread t = new Thread(new
     ThreadedBufferedReaderPrinter(server.getReader()));
t.start();

// start chatting
while (server.isConnected()) {
   try {
      server.writeLine(stdin.readLine());
   } catch (IOException e) {
      e.printStackTrace();
   }
}
```

ThreadedMultiChatServer

- Handle multiple connections with threads
  -
    ```
    while (true)
        accept connection
        start thread to handle connection
    ```
- Multiple clients can connect to the chat server
- Each client managed by a thread, when any client sends a message, bounce to all connected clients
- Store client OutputStreams in a List, all client-handling threads share the list

MultiChatHandler

```java
public class MultiChatHandler implements Runnable {

   public MultiChatHandler(BufferedReader reader,
         List<PrintWriter> outputs, InetAddress addr)
   {
      this.reader = reader;
      this.outputs = outputs;
      name = addr.toString();

      printAll("A new client connected.");
   }

   public void run() {
      while (!Thread.interrupted()) {
         String line = null;
         try {
            line = reader.readLine();
         } catch (IOException e) {
            e.printStackTrace();
         }
         System.out.println(line);
         printAll(line);
      }
   }
```

MultiChatHandler

```
         try {
            line = reader.readLine();
         } catch (IOException e) {
            e.printStackTrace();
         }
         System.out.println(line);
         printAll(line);
      }
   }

   /**
    * Print something to all connected clients
    * @param line
    */
   private void printAll(String line)
   {
      for (PrintWriter pw : outputs)
         pw.println(name + ": " + line);
   }

   BufferedReader reader;
   List<PrintWriter> outputs;
   String name;
}
```

ThreadedMultiChatServer Main Loop

```
List<PrintWriter> allOut = new ArrayList<PrintWriter>();

while(true) {
   try {
     Socket client = server.accept();
     allOut.add(new PrintWriter(client.getOutputStream(), true));
     BufferedReader in = new BufferedReader(
          new InputStreamReader(client.getInputStream()));

     Thread t = new Thread(new
          MultiChatHandler(in, allOut, client.getInetAddress()));
     t.start();
   } catch (IOException e) {
     System.err.println("Error connecting client.");
   }
}
```

Sending Objects Through Streams

- Serialization allows us to send objects through the streams
- Client and Server need to know how to handle the object type
- Harder to debug than sending text, but significant reduction in bandwidth usage
  - also no need for translation code

Binary vs. Text

- An int is 32 bits, a char is 16 bits
- int can represent numbers up to 2147483647 using only 32 bits
- Sending as a String requires 10 chars, 160 bits
- Representing data as its raw binary form saves significant space and time

Serialization Code

- Sending an object:
  - `out = new ObjectOutputStream(socket.getOutputStream());`
  - `out.writeObject(myObject);`
- Receiving object:
  - `in = new ObjectInputStream(socket.getInputStream());`
  - `Object obj = in.readObject(); // or`
  - `MyType obj = (MyType) in.readObject();`

```java
public class RandomListSender {
   private static final int MAX = 10240;

   public static void main(String [] args) {
      Random random = new Random();
      try {
         // open server and create output stream
         ServerSocket server = new ServerSocket(10070);
         Socket socket = server.accept();
         ObjectOutputStream out = new ObjectOutputStream(
               socket.getOutputStream());

         // create the list to send
         List<Integer> list = new LinkedList<Integer>();
         for (int i = 0; i < MAX; i++)
            list.add(random.nextInt());

         out.writeObject(list);

         out.close(); socket.close(); server.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
```

RandomListSender

```
public class ListReceiver {
   public static void main(String [] args) {
      try {
         BufferedReader stdin = new BufferedReader(
               new InputStreamReader(System.in));
         System.out.println("Enter the hostname:");
         String hostname = stdin.readLine();
         System.out.println("Enter the port: ");
         int port = Integer.parseInt(stdin.readLine());

         // open socket
         Socket socket = new Socket(hostname, port);
         ObjectInputStream in = new ObjectInputStream(
               socket.getInputStream());
         // read object from stream
         List<Integer> list =
            (List<Integer>) in.readObject();

         System.out.println(list);

         socket.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
```

ListReceiver

MVC Over the Network

- MVC is commonly used in networked programs where the model and controller are server-side
- Each client has a view of the model, commands are sent to the server, which affect the model
- Model tells all clients to update

Two Patterns in Network Programming

- The Observer pattern fits naturally in network code
  - clients register as observers of data managed by the server
  - The server notifies clients to update
- The Proxy pattern is also a natural fit where objects can be created that represent remote objects locally

Reading

- Horstmann Ch. 10 for Patterns
- http://java.sun.com/docs/books/tutorial/networking/sockets/index.html
- Optional: Ch. 22.1-22.4 in Big Java by Horstmann if you still have it from 1004
- http://www.cs.columbia.edu/~bert/courses/1007/code/networking/