Computer Science II

Professor Tom Ellman
Lecture 20

The Busy Life of a Web Browser

• Recognize and process mouse clicks.
• Download files from the internet.
• Scroll up and down a page.
• Keep those applets running.

Multi-Programming

• Several programs need to run concurrently.
  – E.g., Emacs editor running in one window.
  – E.g., Terminal shell running in one window.
  – E.g., Java program running in another window.
• Only one program can use the CPU at a time.
• Programs need to take turns using the CPU.
• Operating system allocates CPU time-slices to programs.
Multi-Programming

- Linux
- Emacs
- Java
- Terminal

Round-Robin Scheduling

- Run program 1 for one time slice.
- Run program 2 for one time slice.
- Run program 3 for one time slice.
- Resume program 1 for another time slice.
- Resume program 2 for another time slice.
- Resume program 3 for another time slice.
- Etc.

Multi-Threading

- A Java program has several parts that need to run concurrently.
- Each part is implemented as a *thread*.
- Only one thread can use the CPU at a time.
- Threads take turns using the CPU.
- Java virtual machine allocates CPU time-slices to threads.
Threads

- A thread is a single sequential flow of control with a program.
- Each thread has its own program counter and run time stack.
- All threads share access to objects in heap storage.
Defining and Running Threads

- Defining a thread:
  - Define a class `SimpleThread` that extends `Thread`.
  - Provide `SimpleThread` with a `run` method that overrides the `run` method of `Thread`.
- Running a thread:
  - Use the `new` operator to create an instance of `SimpleThread`.
  - Invoke the `start()` method on the newly created `SimpleThread` object.

```java
class SimpleThread extends Thread {
    ConsoleProgram consoleProgram;

    public SimpleThread(ConsoleProgram cProgram, String name) {
        super(name);
        consoleProgram = cProgram;
    }

    // ... Omitted ...
}

public void run() {
    for (int i = 0; i < 10; i++) {
        consoleProgram.println(i + " " + getName());
        try {
            sleep((long)(Math.random() * 1000));
        } catch (InterruptedException e) {
        }
        consoleProgram.println("DONE! " + getName());
    }
}
```
public class SimpleThreadsDemo extends ConsoleProgram {
    public void run() {
        println("Run method creating and starting threads.");
        new SimpleThread(this, "Jamaica").start();
        new SimpleThread(this, "Fiji").start();
        println("Run method is exiting..");
    }
    public static void main(String[] args) {
        new SimpleThreadsDemo().start();
    }
}

Interaction among Threads

- Thread T₁ may call a method of thread T₂.
- The method may modify an instance variable of T₂.
- Thread T₂ may notice that the value of the variable has changed.
- Thread T₂ may then respond accordingly.
Animation Controller Example

- AnimationThreadDemo extends GraphicsProgram which extends Thread.
- The main method of AnimationThreadDemo creates an instance of this class (a Thread) and runs it.
- The AnimationThreadDemo run method creates an instance of ExpandingBall (also extending Thread) and runs it.
- The AnimationThreadDemo thread continues running concurrently with the ExpandingBall thread.
- Java’s event-processing thread waits for mouse clicks and process them by setting instance variables in the ExpandingBall thread.
- The ExpandingBall thread notices the changed variables and responds by starting, stopping or quitting the animation.

```java
public class AnimationThreadDemo extends GraphicsProgram {
    private ExpandingBall expandingBall;
    public void run() {
        addMouseListeners();
        expandingBall = new ExpandingBall(this);
        expandingBall.start();
        final int DELAY = 100;
        while (expandingBall.isAlive())
            try {
                Thread.sleep(DELAY); }
            catch (InterruptedException e) {}
        System.exit(0);
    }
    // ... Omitted ...
    public static void main(String[] args) {
        new AnimationThreadDemo().start();
    }
}
```
public void mousePressed(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    expandingBall.processClick(new GPoint(x,y));
}

This method of AnimationThreadDemo is called by Java's event processing thread whenever the user clicks the mouse in the GraphicsProgram pane.

class ExpandingBall extends Thread {
    private static final int RADIUS = 100;
    private static final int DELAY = 50;
    private static final int WIDTH = 600;
    private static final int HEIGHT = 400;
    private boolean running;
    private boolean quit;
    private GraphicsProgram graphicsProgram;
    private GOval circle;
    Random randomizer = new Random();

    // ... Omitted ...
}

public ExpandingBall(GraphicsProgram gProgram) {
    super("Expanding Ball");
    running = true;
    quit = false;
    graphicsProgram = gProgram;
    circle = new GOval(0,0,2*RADIUS,2*RADIUS);
    switch(Math.abs(randomizer.nextInt())%3) {
    case 0 : circle.setColor(Color.red); break;
    case 1 : circle.setColor(Color.green); break;
    case 2 : circle.setColor(Color.blue); break;
    }
    circle.setFilled(true);
    graphicsProgram.add(circle);
}
public void run() {
    int radius = 0;
    int centerX = 0;
    int centerY = 0;
    while (!quit) {
        if (running) {
            if (radius == 0) {
                int x = Math.abs(randomizer.nextInt()) % (WIDTH - 2*RADIUS);
                int y = Math.abs(randomizer.nextInt()) % (HEIGHT - 2*RADIUS);
                centerX = x + RADIUS;
                centerY = y + RADIUS;
            }
            circle.setBounds(centerX - radius, centerY - radius, 2*radius, 2*radius);
            radius = (radius + 1) % RADIUS;
        }
    }
    try {sleep(DELAY); } catch (InterruptedException e) {}
}

The thread repeatedly executes the body of the while loop as long as quit is false. On each loop body execution the thread expands the ball (if running is true), and then sleeps for a fixed period of time.

public void processClick(GPoint pt) {
    if (circle.contains(pt))
        running = !running;
    else if (circle.getBounds().contains(pt))
        quit = true;
}

The variables quit and running may be changed when the event processing thread calls processClick.
Compare

• Object on which a method is invoked:
  – The `processClick` method is invoked on the `expandingBall` object, which is an instance of the `ExpandingBall` class.

• Thread in which a method runs:
  – The `processClick` method is executed in Java’s event-processing thread, which is not the same thread as the `expandingBall` thread that is running the animation.

Interaction among Threads

• Thread $T_1$ may call a method to `set` the value of an instance variable of an object $S$.

• Thread $T_2$ may call a method to `get` the value of the instance variable of the object $S$.

Producer-Consumer Example

• Producer repeatedly puts data objects in a queue.
  – E.g., Thread downloading an image off the internet puts successive image chunks in a queue.

• Consumer repeatedly gets data objects from a queue.
  – E.g., Thread displaying the image gets successive image chunks out of the queue.
Producer-Consumer Example

Need for Synchronization

• What if producer and consumer run at different speeds?
  – Fast producer might try to put data into a full queue.
  – Fast consumer might try to take data out of an empty queue.
• What if producer and consumer try to access the queue at the same time?
  – Producer or consumer might find the queue in an inconsistent state.

Critical Sections

• A method \( m() \) of class \( C \) may be declared to be “synchronized”.
• When a thread \( T_1 \) calls method \( m() \) on an object \( s \) of class \( C \), the thread \( T_1 \) acquires a lock on object \( s \).
• If another thread \( T_2 \) tries to invoke a synchronized method on object \( s \), thread \( T_2 \) will be blocked.
• When thread \( T_1 \) returns from method \( m() \) on object \( s \), thread \( T_1 \) relinquishes the lock.
• Thread \( T_2 \) will become unblocked and will proceed to run the synchronized method on object \( s \).
Critical Sections Example

1. Thread $T_1$ calls $s.m()$, and locks object $s$.
2. Thread $T_1$ begins running method $s.m()$.
3. Thread $T_2$ calls $s.m()$, and gets blocked.
4. Thread $T_1$ continues running method $s.m()$.
5. Thread $T_1$ returns from method $s.m()$, and unlocks $s$.
6. Thread $T_2$ gets unblocked and proceeds to run $s.m()$.

```java
public class ProducerConsumerDemo extends ConsoleProgram {
    private static final int PRODUCER_DELAY = 500;
    private static final int CONSUMER_DELAY = 1000;

    public void run() {
        int pd = PRODUCER_DELAY;
        int cd = CONSUMER_DELAY;
        Buffer b = new Buffer();
        Producer p = new Producer(this, b, pd);
        Consumer c = new Consumer(this, b, cd);
        p.start();
        c.start();
    }

    public static void main(String[] args) {
        new ProducerConsumerDemo().run();
    }
}
```

```java
public class Producer extends Thread {
    private Buffer buffer;
    private int delay;
    private ConsoleProgram consoleProgram;

    public Producer(ConsoleProgram cProgram, Buffer b, int d) {
        super("Producer");
        consoleProgram = cProgram;
        buffer = b;
        delay = d;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            buffer.put(i);
            consoleProgram.println("Producer put: "+i);
            try { sleep(delay); } catch (InterruptedException e) {
                throw (InterruptedException)e;
            }
        }
    }
}
```
public class Consumer extends Thread {
    private ConsoleProgram consoleProgram;
    private Buffer buffer;
    private int delay;

    public Consumer(ConsoleProgram cProgram, Buffer b, int d) {
        super("Consumer");
        consoleProgram = cProgram;
        buffer = b;
        delay = d;
    }
    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = buffer.get();
            consoleProgram.println("Consumer got: " + value);
            try { sleep(delay); } catch (InterruptedException e) { }
        }
    }
}

public class Buffer {
    private QueueArrayLimited queue;
    private static final int BUFFER_SIZE = 5;

    public Buffer() {
        queue = new QueueArrayLimited(BUFFER_SIZE);
    }
    // ... Omitted ...

    public synchronized void put(int value) {
        while (queue.isFull()) {
            try { wait(); } catch (InterruptedException e) { }
        }
        queue.enQueue(new Integer(value));
        notifyAll();
    }

    When the queue is full, the producer calls \texttt{wait()}. This causes the producer to become blocked and also to relinquish its lock on the buffer. The producer will remain blocked until some other thread (e.g., the consumer) calls \texttt{notifyAll()} on the buffer.
public synchronized int get() {
    while (queue.isEmpty()) {
        try { wait();} catch (InterruptedException e) { }
    }
    int value = queue.dequeue();
    notifyAll();
    return value;
}

When the queue is empty, the consumer calls \texttt{wait()}. This causes the consumer to become blocked and also to relinquish its lock on the buffer. The consumer will remain blocked until some other thread (e.g. the producer) calls \texttt{notifyAll()} on the buffer.