Concurrent Programming: Threads

CS 180
Sunil Prabhakar
Department of Computer Science
Purdue University
Objectives

This week we will get introduced to concurrent programming
- Creating a new thread of execution
- Waiting for threads to terminate
- Thread states and scheduling
- sleep() and yield()
- Simple synchronization among threads
One-track mind?

- Often, in real life we do multiple tasks at the same time
  - Doing the laundry
  - Making a pot of coffee
- This is more efficient.
- Our programs thus far have had a single track (thread) of execution
  - at any point in time, there is only one statement being executed
  - not always efficient -- can stall (e.g., user input)
Multiple concurrent tasks

- Consider a game program that has to repeatedly
  - redraw the scene
  - play the game, record scores, ask the user if they want to play again.
- We don’t want to stop redrawing the scene while waiting for the user input.
- Solution: perform both tasks at the same time (concurrently)
Multiple cores and processors

- Modern computers have multiple CPUs (cores or processors)
- If there is only a single thread of execution, only one CPU is used for our program.
- How do we exploit these other CPUs?
  - Consider
    - the initialization of a large array
    - searching for an item in a large array
  - Split array into pieces and initialize (search) each piece concurrently.
Game: sequential version

```java
initializeGame();
redrawScreen();
boolean done=false;
while(!done) {
    done = processNextMove();
    redrawScreen();
    updateScores();
}
terminateGame();
```

Screen frozen while waiting for user input.
Game: concurrent version

initializeGame();
redrawScreen();

done = processNextMove();
done?
updateScores();

terminateGame();

Note: separate tasks (threads). No freezing
Array: sequential version

```
final int SIZE = 1000000;
double[] rand = new double[SIZE];
for(int i=0; i<SIZE; i++)
    rand[i] = Math.random();
```

Only one thread -- may take long time;
With concurrent execution -- may be twice as fast!

```
int i = 0;
int j = mid;
while (i < mid) {
    rand[i] = Math.random();
    i++
}
while (j < SIZE) {
    rand[j] = Math.random();
    j++
}
```

mid = SIZE/2;

Need to wait for both threads before continuing.
Motivation for concurrency

- Need for asynchrony
  - need to perform separate tasks
    - e.g., game example
  - potential for increased speed with multiple CPUs/cores
    - e.g., matrix example

- Achieving these goals is not straightforward
Concurrent processing

- How do we create a separate thread of execution?
- The **Thread** class provides a facility for creating separate threads.
  - Declare a class to be a descendant of Thread
  - Override the `run()` method to perform the necessary task(s) for the new thread
- When the `start()` method is called, the thread starts executing concurrently
public class Game extends Thread{
    public static void main(String[] args){
        Game game = new Game();
        game.playGame();
    }

    public void playGame(){
        boolean done=false;
        initializeGame();
        start();
        while(!done) {
            done = processNextMove();
            updateScores();
        }
        terminateGame();
    }

    public void run(){
        while(true)
            redrawScreen();
    }
}
public class Game extends Thread{
    public static void main(String[] args){
        Game game = new Game();
        game.playGame();
    }

    public void playGame(){
        boolean done=false;
        initializeGame();
        start();
        while(!done) {
            done = processNextMove();
            updateScores();
        }
        terminateGame();
    }

    public void run(){
        while(true)
            redrawScreen();
    }
}
public class ProcessArray {
    static final int SIZE = 1000000;
    static int[] data = new int[SIZE];

    public static void main(String[] args){
        int mid = SIZE/2;
        InitArray thread1 = new InitArray(0, mid, data);
        InitArray thread2 = new InitArray(mid, SIZE, data);
        thread1.start();
        thread2.start();
    }
}

public class InitArray extends Thread {
    int start, end;
    int array[];
    public InitArray(int from, int to, int[] array){
        start = from;
        end = to;
        this.array = array;
    }
    public void run(){
        for(int i=start;i<end;i++)
            array[i]= Math.random();
    }
}
public class ProcessArray {
    static final int SIZE = 1000000;
    static int[] data = new int[SIZE];

    public static void main(String[] args){
        int mid = SIZE/2;
        InitArray thread1 = new InitArray(0, mid, data);
        InitArray thread2 = new InitArray(mid, SIZE, data);
        thread1.start();
        thread2.start();
        ...
    }
}
**Rejoining threads**

- In the last example, it is necessary to wait for both threads to finish before moving on.
- This is achieved by calling the `join()` method
  - the thread that calls `join` is suspended until the thread on which it is called terminates.
  - this method can throw the (checked) `InterruptedException` so we should catch this exception
public class ProcessArray {
    static final int SIZE = 1000000;
    static int[] odd = new int[SIZE];

    public static void main(String[] args){
        int mid = SIZE/2;
        InitArray thread1 = new InitArray(0, mid);
        InitArray thread2 = new InitArray(mid, SIZE);
        thread1.start();
        thread2.start();

        try{
            thread1.join();
            thread2.join();
        } catch (InterruptedException e){
            System.out.println("Error in thread");
        }
        ...
    }
}

public class InitArray extends Thread {
    int start, end;
    int array[];
    public InitArray(int from, int to, 
                        int[] array){
        start = from;
        end = to;
        this.array = array;
    }
    public void run(){
        for(int i=start;i<end;i++)
            array[i]= Math.random();
    }
}
The `join()` method

- A call to the `join` method blocks (i.e., does not return) until the thread on which it is called terminates
  - returns from its `run()` method, or
  - propagates an exception from `run()`

- While being blocked, the calling thread may get interrupted which is why the `join` method throws the exception.

- Do not use the `stop()` method to stop a thread -- deprecated.
**Speedup**

- Two key reasons for concurrency:
  - liveness (e.g., game keeps redrawing screen)
  - speedup (with more cores, programs run faster)

- Speedup can be measured using the `System` class methods:
  - `public static long currentTimeMillis()`: time elapsed since 1/1/1970 12:00am, in ms
  - `public static long nanoTime()`: current value of computer’s timer in ns.
public class Game extends Thread{
    public static void main(String[] args){
        Game game = new Game();
        game.playGame();
    }

    public void playGame(){
        boolean done=false;
        initializeGame();
        start();
        while(!done) {
            done = processNextMove();
            updateScores();
        }
        terminateGame();
    }

    public void run(){
        while(true)
            redrawScreen();
    }
}
public class ProcessArray {
    static final int SIZE = 1000000;
    static int[] data = new int[SIZE];

    public static void main(String[] args){
        int mid = SIZE/2;
        InitArray thread1 = new InitArray(0, mid, data);
        InitArray thread2 = new InitArray(mid, SIZE, data);
        thread1.start();
        thread2.start();
        ...
    }
}

public class InitArray extends Thread {
    int start, end;
    int array[];
    public InitArray(int from, int to, int[] array){
        start = from;
        end = to;
        this.array = array;
    }
    public void run(){
        for(int i=start;i<end;i++)
            array[i]= Math.random();
    }
}
Creating Sub-Tasks

To achieve concurrent processing, we need to divide a task into multiple pieces that can be assigned to concurrent threads.

Two main approaches

- Task decomposition
  - divide the type of work being performed
    - e.g., game example

- Domain decomposition
  - divide the data on which the same task is performed
    - e.g. matrix initialization
Array multiplication

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\times
\begin{bmatrix}
  b_{11} & b_{12} & b_{13} & b_{14} \\
  b_{21} & b_{22} & b_{23} & b_{24} \\
  b_{31} & b_{32} & b_{33} & b_{34} \\
  b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
= \begin{bmatrix}
  c_{11} & c_{12} & c_{13} & c_{14} \\
  c_{21} & c_{22} & c_{23} & c_{24} \\
  c_{31} & c_{32} & c_{33} & c_{34} \\
  c_{41} & c_{42} & c_{43} & c_{44}
\end{bmatrix}
\]

\[a_{ij} = \sum_{k=1}^{4} b_{ik} \cdot c_{kj} = b_{i1} \cdot c_{1j} + b_{i2} \cdot c_{2j} + b_{i3} \cdot c_{3j} + b_{i4} \cdot c_{4j}\]
Task sub-division

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\begin{bmatrix}
  b_{11} & b_{12} & b_{13} & b_{14} \\
  b_{21} & b_{22} & b_{23} & b_{24} \\
  b_{31} & b_{32} & b_{33} & b_{34} \\
  b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
\times
\begin{bmatrix}
  C_{11} & C_{12} & C_{13} & C_{14} \\
  C_{21} & C_{22} & C_{23} & C_{24} \\
  C_{31} & C_{32} & C_{33} & C_{34} \\
  C_{41} & C_{42} & C_{43} & C_{44}
\end{bmatrix}
\]
public class ArrayMult {
    
    public static void main(String[] args) {
        final int M = 4, N = 4, K = 4;
        int[][] a = new int[M][N];
        int[][] b = new int[M][K];
        int[][] c = new int[K][N];
        // initialize values;
        ArrayMult mult1 = new ArrayMult(0, M / 2, a, b, c);
        ArrayMult mult2 = new ArrayMult(M / 2, M, a, b, c);
        mult1.start();
        mult2.start();
        try {
            mult1.join();
            mult2.join();
        } catch (InterruptedException e) {
            System.out.println("Unexpected Interrupt");
        }
    }
}

public class ArrayMult extends Thread {
    int start, end;
    int[][] a, b, c;

    public ArrayMult(int from, int to, int[][] a, int[][] b, int[][] c) {
        start = from;
        end = to;
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public void run() {
        for (int i = start; i < end; i++) {
            for (int j = 0; j < a[0].length; j++) {
                a[i][j] = 0;
                for (int k = 0; k < b.length; k++)
                    a[i][j] += b[i][k] * c[k][j];
            }
        }
    }
}

Monday, April 2, 2012
Task sub-division

\[
\begin{bmatrix}
    a_{11} & a_{12} & a_{13} & a_{14} \\
    a_{21} & a_{22} & a_{23} & a_{24} \\
    a_{31} & a_{32} & a_{33} & a_{34} \\
    a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
= 
\begin{bmatrix}
    b_{11} & b_{12} & b_{13} & b_{14} \\
    b_{21} & b_{22} & b_{23} & b_{24} \\
    b_{31} & b_{32} & b_{33} & b_{34} \\
    b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
\times
\begin{bmatrix}
    C_{11} & C_{12} & C_{13} & C_{14} \\
    C_{21} & C_{22} & C_{23} & C_{24} \\
    C_{31} & C_{32} & C_{33} & C_{34} \\
    C_{41} & C_{42} & C_{43} & C_{44}
\end{bmatrix}
\]

\[
\begin{bmatrix}
    a_{11} & a_{12} & a_{13} & a_{14} \\
    a_{21} & a_{22} & a_{23} & a_{24} \\
    a_{31} & a_{32} & a_{33} & a_{34} \\
    a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
= 
\begin{bmatrix}
    b_{11} & b_{12} & b_{13} & b_{14} \\
    b_{21} & b_{22} & b_{23} & b_{24} \\
    b_{31} & b_{32} & b_{33} & b_{34} \\
    b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
\times
\begin{bmatrix}
    C_{11} & C_{12} & C_{13} & C_{14} \\
    C_{21} & C_{22} & C_{23} & C_{24} \\
    C_{31} & C_{32} & C_{33} & C_{34} \\
    C_{41} & C_{42} & C_{43} & C_{44}
\end{bmatrix}
\]
```java
public class ArrayMult {
    public static void main(String[] args) {
        final int NUM_THREADS = 5;
        ArrayMult[] threads = new ArrayMult[NUM_THREADS];
        ...

        int subsetSize = (int) Math.ceil(a.length / (float) NUM_THREADS);
        int startRow = 0;
        for (int i = 0; i < NUM_THREADS; i++) {
            threads[i] = new ArrayMult(startRow, Math.min(startRow + subsetSize, a.length), a, b, c);
            threads[i].start();
            start += subsetSize;
        }

        try {
            for (int i = 0; i < NUM_THREADS; i++)
                threads[i].join();
        } catch (InterruptedException e) {
            System.out.println("Unexpected Interrupt");
        }
    }
}
```
Processes

- Modern operating systems support multi-tasking
  - painting the screen, listening to the keyboard, printing, running several programs, ...

- Even with a single processor (single core) multiple tasks are concurrently running

- Achieved by sharing the processor among multiple processes
  - the CPU runs a little of each process in turn
  - this is called process scheduling
Threads

- A process often corresponds to a program
  - Browser, editor, ...
- Modern processes often have multiple threads of execution.
- Roughly,
  - different processes are largely independent of each other;
  - different threads of the same process often share the same memory space.
Thread Scheduling

- Within a single thread, instructions are processed one at a time.
- However, different threads can run at different times/rates.
- When a thread runs is determined by many factors:
  - Java implementation;
  - Operating system
  - Instructions being executed
  - ....
Non-determinism

- An important property of threads is that it is not possible to know exactly when a given thread will be scheduled
  - cannot assume anything about relative ordering between threads (more later)

- Order of concurrent threads (and consequently the result of the output) may change from run to run!

- Programmer must anticipate all possible orderings and protect against possible errors.
Controlling thread scheduling

As a programmer we have several mechanisms available:

- `sleep()`
  - thread cannot be scheduled for some time

- `yield()`
  - voluntarily give up your turn for the CPU

- `wait()`
  - wait for some condition to be true

- Priority
  - Each thread has a priority. Can set priorities for threads we create (with some limitations).
Thread States

- **Runnable**
  - start()
  - yield()
  - (un)schedule
  - run() terminates
  - uncaught exception

- **Running**
  - sleep()
  - wait()
  - notify()
  - wake up

- **Not Runnable**
  - schedule

- **new thread**
Thread scheduling

- At any given time there may be a number of threads that are runnable
  - each has a priority
  - usually the same as the creating thread’s priority

- Periodically, the OS schedules one of the threads with the highest priority for some time.
Synchronization example

Say we want to try to control the relative ordering of two threads:
- thread1 prints: “Left, Left, Left” then “Left”
- thread2 prints: “Right”

Suppose we want to ensure the following output:
- “Left, Left, Left, Right, Left” multiple times.
- How can we ensure that the timing of the threads ensures this output?
- I.e., how to avoid non-determinism?
## Attempt 1: using sleep()

```java
public class LeftThread extends Thread {
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.print("Left ");
            System.out.print("Left ");
            System.out.print("Left ");
            try
            {
                Thread.sleep(50);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
            System.out.println("Left ");
        }
    }
}
```

```java
public class RightThread extends Thread {
    public void run()
    {
        try
        {
            Thread.sleep(10);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        for(int i=0;i<10;i++)
        {
            System.out.print("Right ");
            try
            {
                Thread.sleep(50);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```
Problems with sleep()

- Doesn’t work
  - There is no guarantee that with the sleeping we will get synchronized each time
  - With enough chances, will get out of sync
- There may be unnecessary waiting
- Hard to tune the sleep times
public class LeftThread extends Thread {
    public void run(){
        for(int i=0;i<10;i++){
            System.out.print("Left ");
            System.out.print("Left ");
            System.out.print("Left ");
            Thread.yield();
            System.out.println("Left ");
        }
    }
}

public class RightThread extends Thread {
    public void run(){
        Thread.yield();
        for(int i=0;i<10;i++){
            System.out.print("Right ");
            Thread.yield();
            System.out.println("Left ");
        }
    }
}
Problems with yield()

- Assumes that the yield() calls will give control to the other thread
  - many threads may be running on the machine
  - can cause unexpected switches between our 2 threads
- With multiple cores, each will be running on a separate core -- yielding does not provide anything!
- Also doesn’t work
public class LeftThread extends Thread {
    private RightThread right;
    private boolean done = false;
    public void setRight(RightThread right) {
        this.right = right;
    }
    public void run() {
        for (int i = 0; i < reps; i++) {
            System.out.print("Left ");
            System.out.print("Left ");
            System.out.print("Left ");
            done = true;
            while (!right.isDone());
            right.setDone(false);
            Thread.yield();
            System.out.println("Left ");
        }
    }
    public boolean isDone() { return done; }
    public void setDone(boolean value) { done = value; }
}

public class RightThread extends Thread {
    private LeftThread left;
    private boolean done = false;
    public void setLeft(LeftThread left) {
        this.left = left;
    }
    public void run() {
        for (int i = 0; i < reps; i++) {
            System.out.print("Right ");
            System.out.print("Right ");
            System.out.print("Right ");
            done = true;
            while (!left.isDone());
            left.setDone(false);
            System.out.print("Right ");
            done = true;
        }
    }
    public boolean isDone() { return done; }
    public void setDone(boolean value) {
        done = value;
    }
}
Polling solution?

- This works
  - always produces correct output.

- However,
  - No real concurrency!
  - Only one thread running at a time.
  - Busy waiting (wastes resources)

- Technicality:
  - should ensure that done variables are visible to the other thread immediately: use the **volatile** modifier.
Shared Memory Architecture

- Two common approaches to concurrent programs:
  - message passing
  - shared memory

- Java uses shared memory
  - multiple threads of the same application (program) essentially have access to the same memory space (i.e., variables)
  - memory on each core/CPU is not shared
  - can lead to delays in visibility of modifications (use volatile to avoid these if multiple threads will modify the same variable).
Concurrency is tricky

- Writing concurrent programs that work as expected can be tricky
- Need to deal with
  - non-determinism of scheduling
  - ensuring access to shared data is correct (next week)
- Achieving speed up is not always easy
Examples

- Factorization of a large integer
  - need to find the two prime factors of a large integer value
  - divide the task by domain decomposition

- Array summation
  - compute the sum of the sine of all values of a large array
  - divide by domain decomposition
  - need to synchronize after sub-tasks are done
public class FactorThread extends Thread {
    private long lower;
    private long upper;

    public static final int THREADS = 4;
    public static final long NUMBER = 59984005171248659L;

    public FactorThread(long lower, long upper) {
        this.lower = lower;
        this.upper = upper;
    }

    public void run() {
        if (lower % 2 == 0) {
            lower++;  // Ensure lower is odd if divisible by 2
        }
        while (lower < upper) {
            if (NUMBER % lower == 0) {
                System.out.println("Security Code: "+ (lower + NUMBER/lower));
                return;
            }
            lower += 2;
        }
    }

    public static void main (String[] args) {...}
}
public static void main(String[] args){
    FactorThread[] threads = new FactorThread[THREADS];
    long root = (long)Math.sqrt(NUMBER);
    long start = 3;
    long numbers = (long)Math.ceil((root-2)/(float)THREADS);

    for(int i=0;i<THREADS;i++){
        threads[i] = new FactorThread(start, Math.min(start +numbers, root+1));
        threads[i].start();
        start+=numbers;
    }
    try{
        for(int i=0;i<THREADS;i++)
            threads[i].join();
    } catch (InterruptedException e){
        e.printStackTrace();
    }
}
import java.util.Random;
public class SumThread extends Thread {
    private static double[] data;
    private static SumThread[] threads;
    private double sum = 0;
    private int lower, upper, index;
    public static final int SIZE = 1000000;
    public static final int THREADS = 8;
    public SumThread(int lower, int upper, int index) {
        this.lower = lower;
        this.upper = upper;
        this.index = index;
    }
    public double getSum() { return sum; }
    public void run() { //next slide }
    public static void main(String[] args) { // next slide }
}
public void run () {
    for ( int i = lower ; i < upper ; i++ )
        sum += Math.sin( data [i]);
    int power = 2;
    int neighbor;
    while (index % power == 0 && power < THREADS) {
        neighbor = index + power / 2;
        try { threads [ neighbor ].join (); } 
        catch ( InterruptedException e ) {
            e. printStackTrace ();
        }
        sum += threads [ neighbor ].getSum ();
        power *= 2;
    }
}
...

Monday, April 2, 2012
```java
public static void main ( String [] args ) {
    data = new double [ SIZE ];
    Random random = new Random ();
    int start = 0;
    for ( int i = 0; i < SIZE ; i++ )
        data [i] = random . nextDouble ();
    threads = new SumThread [ THREADS ];
    int range = ( int ) Math . ceil ( data . length / ( float ) THREADS );
    for ( int i = 0; i < THREADS ; i++ ) {
        threads [i] = new SumThread ( start, Math . min( start + range , SIZE ), i );
        threads [i]. start ();
        start += range ;
    }
    try { threads [0]. join (); }
    catch ( InterruptedException e ) {
        e. printStackTrace ();
    }
    System .out. println ("Sum: " + threads [0]. getSum ());
}
```