Principles of Software Construction: Objects, Design, and Concurrency

Part 5: Concurrency

Introduction to concurrency, part 4

Concurrency frameworks

Charlie Garrod    Michael Hilton
Administrivia

- Homework 5b due tonight 11:59 p.m.
  - Turn in by Wednesday 9 a.m. to be considered as a Best Framework
Key concepts from last Thursday
Summary of our RwLock example

- Generally, avoid `wait/notify`
- Never invoke `wait` outside a loop
  - Must check coordination condition after waking
- Generally use `notifyAll`, not `notify`
- Do not use our RwLock – it's just a toy
  - Instead, know the standard libraries...
    - Discuss: `sun.misc.Unsafe`
Concurrent bugs can be very subtle

```java
private final List<Observer<E>> observers = new ArrayList<>();
public void addObserver(Observer<E> observer) {
    synchronized(observers) { observers.add(observer); }
}
public boolean removeObserver(Observer<E> observer) {
    synchronized(observers) { return observers.remove(observer); }
}
private void notifyOf(E element) {
    synchronized(observers) {
        for (Observer<E> observer : observers)
            observer.notify(this, element); // Risks liveness and
                                    // safety failures!
    }
}
```
The fork-join pattern

if (my portion of the work is small)
do the work directly
else
    split my work into pieces
    invoke the pieces and wait for the results
The membrane pattern

- Multiple rounds of fork-join, each round waiting for the previous round to complete
Today

• An aside: Networking in Java
• The Java executors framework
• Concurrency in practice: In the trenches of parallelism
Basic types in Java

• What is a byte?
  – Answer: a signed, 8-bit integer (-128 to 127)

• What is a char?
  – Answer: a 16-bit Unicode-encoded character
The stream abstraction

- A sequence of **bytes**
- May read 8 bits at a time, and close

```java
java.io.InputStream
    void close();
    abstract int read();
    int read(byte[] b);
```

- May write, flush and close

```java
java.io.OutputStream
    void close();
    void flush();
    abstract void write(int b);
    void write(byte[] b);
```
Example streams

• `java.io.FileInputStream`
  – Reads from files, byte by byte
• `java.io.ByteArrayInputStream`
  – Provides a stream interface for a byte[]
• Many APIs provide streams
  – e.g., `java.lang.System.in`
Aside: To read and write arbitrary objects

• Your object must implement the java.io.Serializable interface
  – Methods: none

• If all of your data fields are themselves Serializable, Java can automatically serialize your class
  – If not, will get runtime NotSerializableException

• Can customize serialization by overriding special methods
Internet addresses and sockets

- For IP version 4 (IPv4) host address is a 4-byte number
  - e.g. 127.0.0.1
  - Hostnames mapped to host IP addresses via DNS
  - ~4 billion distinct addresses

- Port is a 16-bit number (0-65535)
  - Assigned conventionally
    - e.g., port 80 is the standard port for web servers
Packet-oriented and stream-oriented connections

- **UDP**: User Datagram Protocol
  - Unreliable, discrete packets of data
- **TCP**: Transmission Control Protocol
  - Reliable data stream
Networking in Java

• The java.net.InetAddress:
  static InetAddress getByName(String host);
  static InetAddress getByAddress(byte[] b);
  static InetAddress getLocalHost();

• The java.net.Socket:
  Socket(InetAddress addr, int port);
  boolean isConnected();
  boolean isClosed();
  void close();
  InputStream getInputStream();
  OutputStream getOutputStream();

• The java.net.ServerSocket:
  ServerSocket(int port);
  Socket accept();
  void close();
  ...

Today

• An aside: Networking in Java
• The Java executors framework
• Concurrency in practice: In the trenches of parallelism
Execution of tasks

• Natural boundaries of computation define tasks, e.g.:

```java
public class SingleThreadWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }

    private static void handleRequest(Socket connection) { 
        ... // request-handling logic here
    }
}
```
A poor design choice: A thread per task

public class ThreadPerRequestWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            new Thread(() -> handleRequest(connection)).start();
        }
    }
}

private static void handleRequest(Socket connection) {
    ... // request-handling logic here
}
Recall the Java primitive concurrency tools

- The `java.lang.Runnable` interface
  ```java
  void run();
  ```

- The `java.lang.Thread` class
  ```java
  Thread(Runnable r);
  void start();
  void join();
  ```
Recall the Java primitive concurrency tools

- The `java.lang.Runnable` interface
  ```java
  void run();
  ```

- The `java.lang.Thread` class
  ```java
  Thread(Runnable r);
  void start();
  void join();
  ```

- The `java.util.concurrent.Callable<V>` interface
  - Like `java.lang.Runnable` but can return a value
  ```java
  V call();
  ```
A framework for asynchronous computation

- The `java.util.concurrent.Future<V>` interface
  
  ```java
  V get();
  V get(long timeout, TimeUnit unit);
  boolean isDone();
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  ```
A framework for asynchronous computation

- The `java.util.concurrent.Future<V>` interface:
  
  ```java
  V get();
  V get(long timeout, TimeUnit unit);
  boolean isDone();
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  ```

- The `java.util.concurrent.ExecutorService` interface:
  
  ```java
  Future<?> submit(Runnable task);
  Future<V> submit(Callable<V> task);
  List<Future<V>> invokeAll(Collection<? extends Callable<V>> tasks);
  Future<V> invokeAny(Collection<? extends Callable<V>> tasks);
  void shutdown();
  ```
Executors for common computational patterns

• From the java.util.concurrent.Executors class
  static ExecutorService newSingleThreadExecutor();
  static ExecutorService newFixedThreadPool(int n);
  static ExecutorService newCachedThreadPool();
  static ExecutorService newScheduledThreadPool(int n);
Example use of executor service

```java
public class ThreadPoolWebServer {
    private static final Executor exec = Executors.newFixedThreadPool(100); // 100 threads

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            exec.execute(() -> handleRequest(connection));
        }
    }

    private static void handleRequest(Socket connection) {
        ... // request-handling logic here
    }
}
```
Today

- An aside: Networking in Java
- The Java executors framework
- Concurrency in practice: In the trenches of parallelism
Concurrency at the language level

• Consider:

```java
Collection<Integer> collection = ...;
int sum = 0;
for (int i : collection) {
    sum += i;
}
```

• In python:

```python
collection = ...
sum = 0
for item in collection:
    sum += item
```
Parallel quicksort in Nesl

function quicksort(a) =
  if (#a < 2) then a
  else
    let pivot = a[#a/2];
    lesser = {e in a| e < pivot};
    equal = {e in a| e == pivot};
    greater = {e in a| e > pivot};
    result = {quicksort(v): v in [lesser,greater]};
    in result[0] ++ equal ++ result[1];

• Operations in {} occur in parallel
• 210-esque questions: What is total work? What is depth?
Prefix sums (a.k.a. inclusive scan, a.k.a. scan)

• Goal: given array $x[0...n-1]$, compute array of the sum of each prefix of $x$
  
  $\begin{array}{l}
    \text{sum}(x[0...0]), \\
    \text{sum}(x[0...1]), \\
    \text{sum}(x[0...2]), \\
    \vdots \\
    \text{sum}(x[0...n-1]) \\
  \end{array}$

• e.g., $x = [13, 9, -4, 19, -6, 2, 6, 3]$
  
  prefix sums: $[13, 22, 18, 37, 31, 33, 39, 42]$
Parallel prefix sums

• Intuition: If we have already computed the partial sums $\text{sum}(x[0...3])$ and $\text{sum}(x[4...7])$, then we can easily compute $\text{sum}(x[0...7])$

• e.g., $x = [13, 9, -4, 19, -6, 2, 6, 3]$
Parallel prefix sums algorithm, **upsweep**

Compute the partial sums in a more useful manner

\[
\begin{bmatrix}
13, & 9, & -4, & 19, & -6, & 2, & 6, & 3 \\
13, & 22, & -4, & 15, & -6, & -4, & 6, & 9
\end{bmatrix}
\]
Parallel prefix sums algorithm, \textbf{upsweep}

Compute the partial sums in a more useful manner

\begin{align*}
[13, & 9, -4, 19, -6, 2, 6, 3] \\
[13, & 22, -4, 15, -6, -4, 6, 9] \\
[13, & 22, -4, 37, -6, -4, 6, 5]
\end{align*}
Parallel prefix sums algorithm, \textit{upsweep}

Compute the partial sums in a more useful manner

\begin{align*}
[13, & \quad 9, \quad -4, \quad 19, \quad -6, \quad 2, \quad 6, \quad 3] \\
[13, & \quad 22, \quad -4, \quad 15, \quad -6, \quad -4, \quad 6, \quad 9] \\
[13, & \quad 22, \quad -4, \quad 37, \quad -6, \quad -4, \quad 6, \quad 5] \\
[13, & \quad 22, \quad -4, \quad 37, \quad -6, \quad -4, \quad 6, \quad 42]
\end{align*}
Parallel prefix sums algorithm, **downsweep**

Now unwind to calculate the other sums

\[
[13, 22, -4, 37, -6, -4, 6, 42]\]

\[
[13, 22, -4, 37, -6, 33, 6, 42]\]
Parallel prefix sums algorithm, **downsweep**

Now unwind to calculate the other sums

\[
\begin{bmatrix}
13, & 22, & -4, & 37, & -6, & -4, & 6, & 42 \\
13, & 22, & -4, & 37, & -6, & 33, & 6, & 42 \\
13, & 22, & 18, & 37, & 31, & 33, & 39, & 42 \\
\end{bmatrix}
\]

- Recall, we started with:

\[
\begin{bmatrix}
13, & 9, & -4, & 19, & -6, & 2, & 6, & 3 \\
\end{bmatrix}
\]
Doubling array size adds two more levels
Parallel prefix sums

**pseudocode**

// Upsweep
prefix_sums(x):
    for d in 0 to (lg n)-1:  // d is depth
        parallel for i in 2^d-1 to n-1, by 2^{d+1}:
            x[i+2^d] = x[i] + x[i+2^d]

// Downsweep
for d in (lg n)-1 to 0:
    parallel for i in 2^d-1 to n-1-2^d, by 2^{d+1}:
        if (i-2^d >= 0):
            x[i] = x[i] + x[i-2^d]
Parallel prefix sums algorithm, in code

• An iterative Java-esque implementation:

```java
void iterativePrefixSums(long[] a) {
    int gap = 1;
    for (; gap < a.length; gap *= 2) {
        parfor(int i=gap-1; i+gap < a.length; i += 2*gap) {
            a[i+gap] = a[i] + a[i+gap];
        }
    }
    for (; gap > 0; gap /= 2) {
        parfor(int i=gap-1; i < a.length; i += 2*gap) {
            a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
        }
    }
}
```
Parallel prefix sums algorithm, in code

- A recursive Java-esque implementation:

```java
void recursivePrefixSums(long[] a, int gap) {
    if (2*gap - 1 >= a.length) {
        return;
    }

    parfor(int i=gap-1; i+gap < a.length; i += 2*gap) {
        a[i+gap] = a[i] + a[i+gap];
    }

    recursivePrefixSums(a, gap*2);

    parfor(int i=gap-1; i < a.length; i += 2*gap) {
        a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
    }
}
```
Parallel prefix sums algorithm

• How good is this?
Parallel prefix sums algorithm

- How good is this?
  - Work: $O(n)$
  - Depth: $O(\lg n)$

- See PrefixSums.java, PrefixSumsSequentialWithParallelWork.java
Goal: parallelize the PrefixSums implementation

- Specifically, parallelize the parallelizable loops
  ```java
  parfor(int i = gap-1; i+gap < a.length; i += 2*gap) {
    a[i+gap] = a[i] + a[i+gap];
  }
  ```
- Partition into multiple segments, run in different threads
  ```java
  for(int i = left+gap-1; i+gap < right; i += 2*gap) {
    a[i+gap] = a[i] + a[i+gap];
  }
  ```
Recall: The membrane pattern

- Multiple rounds of fork-join, each round waiting for the previous round to complete

![Diagram of membrane pattern with parallel tasks](Image from: Wikipedia)
Fork/join in Java

• The java.util.concurrent.ForkJoinPool class
  – Implements ExecutorService
  – Executes java.util.concurrent.ForkJoinTask<V> or java.util.concurrent.RecursiveTask<V> or java.util.concurrent.RecursiveAction

• In a long computation:
  – Fork a thread (or more) to do some work
  – Join the thread(s) to obtain the result of the work
The RecursiveAction abstract class

public class MyActionFoo extends RecursiveAction {
    public MyActionFoo(...) {
        store the data fields we need
    }

    @Override
    public void compute() {
        if (the task is small) {
            do the work here;
            return;
        }

        invokeAll(new MyActionFoo(...), // smaller
                   new MyActionFoo(...), // subtasks
                   ...); // ...
    }
}
A ForkJoin example

• See PrefixSumsParallelForkJoin.java
• See the processor go, go go!
Parallel prefix sums algorithm

• How good is this?
  – Work: $O(n)$
  – Depth: $O(\lg n)$

• See PrefixSumsParallelArrays.java
Parallel prefix sums algorithm

• How good is this?
  – Work: $O(n)$
  – Depth: $O(\lg n)$

• See PrefixSumsParallelArrays.java
• See PrefixSumsSequential.java
Parallel prefix sums algorithm

• How good is this?
  – Work: $O(n)$
  – Depth: $O(\lg n)$

• See PrefixSumsParallelArrays.java

• See PrefixSumsSequential.java
  – n-1 additions
  – Memory access is sequential

• For PrefixSumsSequentialWithParallelWork.java
  – About $2n$ useful additions, plus extra additions for the loop indexes
  – Memory access is non-sequential

• The punchline:
  – Don't roll your own
  – Cache and constants matter
In-class example for parallel prefix sums

\[ [7, 5, 8, -36, 17, 2, 21, 18] \]