SE 552: Lecture 10

Overview

Services in threads

Parallel decomposition
Services in Threads

Problem: how do we get a result back from an asynchronous method call?

First: why do we want to? Isn’t the point of an asynchronous method call that we don’t wait for a result?

Example pseudocode:

```java
void newDocument (final String filename) throws IOException {
    final FileInputStream in = new FileInputStream (filename);
    final Document document =
        Document.factory .build (in); // slow because of disk I/O
    final DocumentFrame frame =
        DocumentFrame.factory .build (); // slow because of graphics I/O
    frame.show (document);
}
```

How could we use threads to improve the performance of this code?
Services in threads

We would like to implement something like:

```java
foo.asyncMethodCall();
execute other code
final Result result = result of foo.asyncMethodCall();
do something with the result
```

This is called *deferred synchronous invocation*.

Possible implementations: callbacks, thread joining, futures.
Callbacks

Implement:

```java
foo.asyncMethodCall ();
execute other code
final Result result = result of foo.asyncMethodCall ();
do something with the result
```

using:

```java
final Callback callback = new Callback () {
    public void acceptResult (final Result result) {
        do something with the result
    }
}
foo.asyncMethodCall (callback); // accepts a callback now
execute other code
```

Can we program the newDocument method this way?
Thread joining

Implement:

```java
foo.asyncMethodCall ();
execute other code
final Result result = result of foo.asyncMethodCall ();
do something with the result
```

using:

```java
class ResultThread extends Thread () { public Result result = null; }
final ResultThread thread = new ResultThread () { public void run () {
    this.result = foo.syncMethodCall (); // call the sync. version here
}
};
thread.start ();
execute other code
thread.join ();
final Result result = thread.result;
do something with the result
```

Can we program the newDocument method this way?
Futures

Implement:

```java
foo.asyncMethodCall ();
execute other code
final Result result = result of foo.asyncMethodCall ();
do something with the result
```

using:

```java
FutureResult future = foo.asyncMethodCall (); // returns a future
execute other code
final Result result = future.get ();
do something with the result
```

Can we program the newDocument method this way?

This is an example of using an IVar.
Parallel decomposition

Sometimes we want to do *parallel decomposition* of programs for a SMP architecture. General pattern of divide-and-conquer:

```java
pseudoclass Solver {
  ...
  Result solve (final Problem p) {
    if (problem.size < BASE_CASE_SIZE) {
      return directlySolve (p);
    } else {
      final Result l = solve (lefthalf (p));
      final Result r = solve (righthalf (p));
      return combine (l, r);
    }
  }
}
```

How can we sort a list using this pattern? What are other examples of divide-and-conquer?

How can we exploit multiple processors with this pattern?
A framework for parallel divide-and-conquer

A framework for parallel divide-and-conquer using *Fork and Join* primitives:

```java
abstract class FJTask implements Runnable {
    void fork () { ... }
    void join () { ... }
    ...
}
```
A framework for parallel divide-and-conquer

An example of using the framework:

class Fibonacci extends FJTask {
    final int input;
    volatile int output = -1;
    Fibonacci (final int input) { this.input = input; }
    public void run () {
        if (input <= 1) { this.output = 1; }
        else {
            final Fibonacci task1 = new Fibonacci (input-1);
            final Fibonacci task2 = new Fibonacci (input-2);
            task1.fork (); task2.fork ();
            task1.join (); task2.join ();
            this.output = task1.output + task2.output;
        }
    }
}

How does this work? How can we implement FJTask?
Computation trees

The fork-join mechanism deals well with divide-and-conquer algorithms, but not every algorithm is divide-and-conquer.

Many algorithms look like:

```java
while (some condition) {
    perform a divide-and-conquer operation
}
```

Example: how could we (very naively) compute \( \text{fib}(n) + \text{fib}(n) + \ldots + \text{fib}(n) \) (\( m \) times)?

We could just implement these using ForkJoin. Why might this be inefficient?

Instead, we could precompute the *tree structure* of the divide-and-conquer calls, and use the same tree each time round the loop.
Summary

Asynchronous method calls are fine and good, but sometimes we need to get data back from a method call! Possibilities include: callbacks, thread joining and futures (aka IVars).

For SMP machines, we can exploit parallelism by using a fork-and-join strategy for parallelising computation. Simple algorithms for this include divide-and-conquer and computation trees.

Next week: Final exam.