JThreadSpy

Teaching Multithreading Programming by Analyzing Execution Traces

Giovanni Malnati, Caterina Maria Cuva, Claudia Barberis

Dipartimento di Automatica e Informatica
Politecnico di Torino
Teaching multi-threading: needs

- Teaching multi-threaded programming is a difficult task
  - Synchronization problems are often presented only at a very abstract level
  - Students have to figure out what happens in their programs
    - Single-threaded programs analysis and debugging techniques are not useful
    - Subsequent executions of the same program can produce different execution flows
      - Intrinsic non-determinism of thread scheduling
JThreadSpy: goals

- JThreadSpy is an educational tool
  - Aimed at improving students consciousness of race conditions and multithreading issues
  - Useful to detect anomalies in concurrent programs

- Traces are collected during program execution
  - Registering relevant events
  - A code instrumentation technique is used
  - Execution flows are graphically displayed
    - Synchronization constructs are shown
Collecting execution traces

- Manually insert instructions in the source code
  - Expensive and error prone approach
- Replace the JVM with a custom one
  - Overwhelming task
- Use information provided by the Java Virtual Machine Tool Interface
  - Not available in all JVM implementations
  - Not portable across different operating systems
- Use aspect-oriented programming
  - Easy to use
  - Need of installing the runtime environment
  - Need of some notion of aspects
    - Not present in every curriculum
- Dynamically instrument bytecode
Dynamic bytecode instrumentation

- At class load time, bytecode is inspected by a Java agent registered during the JVM start-up
  - Extra instructions are inserted in order to flag method call and return and relevant synchronization constructs
  - When executed, the inserted code will register the corresponding event in a shared list which will be later saved to a file for subsequent inspection

- Advantages
  - No modification is needed to source code
  - Very low latency between the event and its recording, although some overhead is introduced

- Drawbacks
  - Java agent supported only since JVM version 1.5
  - Core Java classes cannot be instrumented on the fly
JThreadSpy architecture

- Java agent
  - Instruments classes and produces a trace file
- Visualizer
  - Decodes the trace file and produces an enhanced UML sequence diagram
- Eclipse plug-in
  - Integrates the above components inside the IDE
Two categories of events are logged

- Method invocation
  - Object instance’s methods
  - Constructors
  - Static methods

- Synchronization constructs
  - Synchronized methods
  - Synchronized blocks of code
  - Object.wait()
  - Object.notify() e Object.notifyAll()
Each event record consists of

- An event type
- A unique thread identifier
- A unique object identifier
- The class name of the object
- The class name of the caller object
- The name of the traced method
- The current stack depth
- Three timestamps
  - Event start
  - Critical section acquisition (for synchronization events)
  - Event end
Every method is replaced by a stub

- Original bytecode is inserted in new private methods
- “[hidden]” is used to prefix method names

The actual method is called in a try/finally block

- An event is created before method invocation, with the initial timestamp
- The finally code updates the event with the return timestamp

```java
<anyVisibility> <anyReturnType>
methodName(...) {
    //create trace event
    TraceEvent te =
        new TraceEvent(this, ...);
    try {
        return [hidden]methodName(...);
    } finally {
        //update & store trace event
        Reporter.exiting(te);
    }
}

private <anyReturnType>
[hidden]methodName(...) {
    //original code modified
    //in order to monitor
    //access to critical sections
}
```
The previous approach is not suitable for constructors

- An overloaded version of the constructor is introduced
- For constructors, the event is created without an object identifier
  - It is set only when the constructor returns
  - If an exception is raised during the super-class construction, the corresponding event is discarded

```java
<anyVisibility> ClassName(...) {  
    this(...,
        new TraceEvent(null, ...));  
}  
private ClassName(..., TraceEvent te) {  
    super(...);
    try {  
        //set trace event object id  
        //original code modified  
        //in order to monitor  
        //access to critical sections  
        //before each return instruction,  
        //the trace event is updated  
        //with the proper timestamp  
    } catch(Throwable t) {  
        //update & store trace event  
    }  
}  
```
Synchronized methods need to acquire a lock in order to actually start
✓ Same result of executing their code within a synchronized block of code

Code is rewritten similarly to other methods
✓ New private methods are not synchronized
✓ They are called within a synchronized block of code
✓ The lock acquisition timestamp is set before calling the method

```java
<anyVisibility> <anyReturnType>
  methodName(...) {
    TraceEvent te =
      new TraceEvent(this, ...);
    try {
      synchronized(this) {
        // update acquisition time
        // in the trace event
        return [hidden]methodName(...);
      }
    } finally {
      // update & store trace event
      Reporter.exiting(te);
    }
  }

private <anyReturnType>
  [hidden]methodName(...) {
    // original code modified
    // in order to monitor
    // access to critical sections
  }
```
Code rewriting rules (III)

- Synchronized blocks of code are enclosed within a `try/finally` block
  - An event is created before synchronized block
    - Contains the start event timestamp
  - The lock acquisition timestamp is set before the first instruction of the synchronized block
  - The code in the finally block sets the release timestamp
- `wait()` instructions are enclosed within a `try/finally` block
  - An event is created before `wait()` instruction
    - Contains the start event timestamp
  - The finally code sets the end timestamp
- `notify()` and `notifyAll()` instructions
  - A single timestamp is recorded
A shutdown hook is registered

- Launched when the JVM shutdown starts
- Sets end timestamps of events not yet terminated
  - Useful in case of System.exit() invocation
- Serializes events into a file for the subsequent visualization
- Standard notation is used where available
  - Object instances
  - Lifelines
  - Activation lines
  - Method invocations
  - Timeline
    - Starts from the top of the diagram
    - Time increases downwards
Augmented UML notation (I)

- Static method invocation
  - Related to the class of the method
  - Represented with a rounded rectangle
  - Other rules as for standard objects

- One trace for each thread
  - Each thread is drawn with a different colour
Sometimes, methods are called from non-instrumented code

- Represented with a broken horizontal arrow
- A wave-shaped vertical line depicts execution taking place in non-instrumented code
Augmented UML notation (III)

- **Synchronization**
  - A padlock identifies
    - Synchronized method calls
    - Synchronized blocks of code
  - Synchronized blocks of code are marked with dotted start and end arrows
  - An hourglass identifies wait instructions
  - An exclamation mark is used for notify instructions
    - `notifyAll` is represented by a double exclamation mark
  - A dotted and dashed line on the left of the lifeline identifies a thread waiting for lock acquisition
  - A semi-transparent rectangle overlapping the lifeline shows that the thread owns the object lock
Augmented UML notation (IV)
Results

- First student feedbacks are generally positive
  - The tool helps in creating a visual representation of the execution of programs
  - It stimulates personal experimentation and it highlights several details of the inner working of the JVM, which are often disregarded in OO courses
  - People tend to engage more and to contribute to the improvement of the tool with useful suggestions

- Still a lot of work to be done
  - Currently tested only with a small “controlled” group (15 people)
  - Next year it will be used with the whole class (about 150 students)
Current issues and future work

- Enhance the code rewrite engine
  - Provide support for the semantics of the Java Synchronization Framework classes and interfaces
  - Trace access to object fields
- Introduce reasoning about collected data
  - Highlight possible conflicts
  - Identify blocked threads
  - Provide suggestions about possible anti-patterns
- Improve the usability of the visualizer
  - Print information about the pointed object
  - Introduce a navigation modality, in order to follow the execution of a given thread, automatically scrolling back and forth
  - Support dynamic object layout rearrangement
JThreadSpy in action
Thank you!

giovanni.malnati@polito.it
caterina.cuva@polito.it