ICP Notes

Kyle Kroboth

October 29, 2017

Disclaimer
This document is a work in progress and contains notes that may be confusing or incomplete.

Contents

1 Notes

1.1 Constructs skipped during class editing

1. Packages: `javaassist.*`, `icp.core`, `sbt.`, `org.testng.`, `com.intelij.rt.`
2. `static` and `abstract` methods are not instrumented.
3. No permission checks on `final` and `static` fields.
4. Permission field not added on: `interfaces`, `@External Annotation`

1.2 Lambdas are thread-safe

Anytime a Lambda is used, it will always be thread-safe. By default (TODO...)

1.3 Synchronizers

All synchronizers should be **`final`**

2 Ideas and Research

Features that are in consideration.

2.1 ICPCollections

Just how `Collections.newXXX` static methods, there could be similar style for wrapping java.util maps, lists, and more. Other possibility is creating individual wrapper classes.

2.2 Use JavaAgent

JavaAgent's allow setting of custom class loader among other inspections. See https://zeroturnaround.com/rebell
to-inspect-classes-in-your-jvm/

2.3 Final Fields

Look into how JLS handles static vs non-static compiler inlines. See if a static call to `getClass()` before access to final static field is made.

2.4 Replace Monitor locks with Reentrant instance locks

Without digging into the JVM, we cannot correctly check permissions for which a shared operation is accessed with a synchronized block or method.

One possibility is replacing `MonitorEnter` and `MonitorExit` bytecode ops with lock() and unlock() methods of a reentrant lock. Depends on how Monitors are used.

```
public synchronized void syncMethod() {
    shared++;
}
```

Listing 1: Synchornized Method

```
public synchronized void syncMethod();
    Code:
        0: aload_0
        1: dup
        2: getfield      #2                  // Field shared:I
        5: iconst_1
        6: iadd
        7: putfield      #2                  // Field shared:I
        10: return
```

Listing 2: Bytecode for Listings 1

In the cases of Listings 1 and 2, no bytecode ops for monitor enters and exists are used. Instead the JVM checks method for synchronized flag.

```
public void syncBlock() {
    synchronized(this) {
        shared++;
    }
}
```

```
public void syncBlock();
    Code:
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: aload_0
       5: dup
       6: getfield      #2                  // Field shared:I
       9: iconst_1
      10: iadd
      11: putfield      #2                  // Field shared:I
      14: aload_1
      15: monitorexit
      16: goto          24
      19: astore_2
      20: aload_1
      21: monitorexit
      22: aload_2
      23: athrow
      24: return
    Exception table:
       from    to  target type
           4    16    19   any
          19    22    19   any
```

Explicit monitor enter and exit are used for synchronized blocks.

One strategy for synchronized blocks is replacing a the enter and exit ops with a reentrant lock. Doing so gives ICP full control and allow the user to use permissions associated with object monitors. The problem lies in how the system finds and replaces the bytecode.

2.4.1 Possible solution

Since it may be difficult to know which object's monitor is used, inserting support functions before `monitorenter` and after `aload_0` (in these cases) may be able to get the object. Remove the monitor bytecode, insert reentrant lock and unlock on matching `monitorexit`. Must use exception tables to always unlock lock.

3 Problems

3.1 Use of @External Annotation

The External annotation is used to mark classes which should not have permission fields injected. Use cases may include external libraries[1], test classes and stateless static method classes.

The problem arises when External classes are used in inheritance chains. Consider the following inheritance chain:
`Object <- A <- B (External) <- C`

Before moving on to scenarios, our class editing implementation[2] for adding permission fields skips under these conditions:

- Class has `@External` annotation
- Permission field exists in superclass

Whenever `ICP.setPermission(<object>, <permission>)` is called, a lookup for the permission field on `<object>` is performed. That is done by the following: This algorithm returns the field closest up the inheritance chain and does not care if a class is @External.

Finally, we have no control over how JavaAssist class loader loads classes. At any time, class A, B, or C can be loaded.

4 Synchronizers with Registrations

Registration refers to tasks saying *I'm going to perform this operation* before calling that operation. For example, for the case of 1-time latches, a Task will inform the synchronizer it

[1] Right now there is no way to exclude entire or include specific packages
[2] See PermissionSupport.addPermissionField()

is a *opener* before calling `oneTimeLatch.open()`. This allows a disjoint set of opener and waiter tasks.

As a side-effect of registration, are possible. One-time latch permissions may include *IsOpenPermission* and *IsClosedPermission*. To use these permissions, three objects must be used: *Data*[3], *Accessor*, and *Provider*. This makes coding awkward.

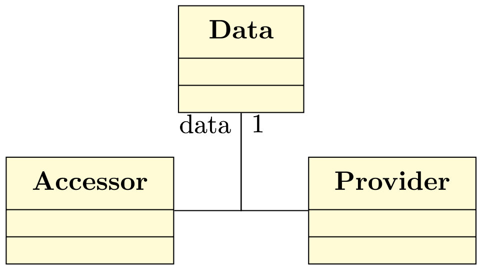
```
    // Shared operation (thread safe)
    static class Data {
      int counter = 0;

      Data() {
        ICP.setPermission(this, Permissions.
 getPermanentlyThreadSafePermission());
      }
    }

    // Worker accesses (thread safe)
    static class Provider {
      final Data data;

      Provider(Data data) {
        this.data = data;
        ICP.setPermission(this, Permissions.
 getPermanentlyThreadSafePermission());
      }

      void setData(int counter) {
        this.data.counter = counter;
      }
    }

    // Accessor (is open permission)
    static class Accessor {
      final Data data;

      Accessor(Data data) {
        this.data = data;
      }

      int getCounter() {
        return data.counter;
      }
    }
```

5 Task-based permissions

The original intent of this project was to tie clear permissions to objects. Typically, only one invariant exists per Permission. *IsOpen*, *IsClosed*, *HasLock* and so on.

[3]Usually thread-safe

The downside is there can only be one permission for a object. A simple workaround is having the user create three classes: Underlying shared data, Accessor, and Provider. Then each object has a unique permission attached. For latches, *IsOpen* goes to Accessor, and *IsClosed* for Provider. This leaves the data object to be thread-safe in some cases.



5.1 Failed Solution: Compound Permissions

To get around multiple permissions on a single object, a *Compound* permission that uses composition of permissions. `new JoinPermission(new ClosedPermission());` [4] Too complicated to implement and gets worse after composing more than one permission.

5.2 Solution: Registration

By using Boolean TaskLocal's in the synchronizer, it is possible to have one [5] permission tied to one object which has multiple invariants depending on the current task. The common strategy is:

1. Task tells synchronizer it will call an operation
2. Task then calls operation

Simple, but can be cumbersome for the user:

```
    // Worker task doing some work then calling countDown on
CountDownLatch
    // ... work
    latch.countDowner(); // Worker says it's a countdowner
    latch.countDown(); // Now worker may call countDown
```

The mistakes being caught are non-countdowners calling countdown operation; Master should not wait and call countdown. Though the example has the two methods right next to each other, it is usually writen as:

```
    CountDownLatch latch = new CountDownLatch(10); // ICP countdown
latch of 10
    SharedData data = new SharedData();
    ICP.setPermission(data, latch.getPermission());
```

[4] A failed attempt of Task-Joining permission. Ideally, once a worker task joins, the master is able to see any shared objects the worker had (JMM). But, the worker task cannot acesses shared data after opening the latch while it is running.

[5] *The Permission* to rule them all. For example, 1-time latch with registration has only one permission for both IsClosed and IsOpen checks.

```
        for(int i = 0; i < 10; i++) {
            new Thread(Task.newThreadSafeRunnable(() -> {
                // Worker registers as countdowner in first line of
    runnable (task)
                latch.countDowner();
                // ... more initialization

                // ... calculations

                data.set(i, <worker data>);
                latch.countDown(); // valid, worker is registered
            })).start();
        }

        latch.await();
        // ... Access data
```

The goal is having the user register their worker tasks immediately in task initialization. It would still be valid if `latch.countDowner()` was called right before the count down.

6 Three Object Idiom

Permissions are tied at the Object level and not method.[6] Take a simple Data object that uses a AtomicInteger as its underlying data. *Provider* increments the integer, while *Accessor* retrieves the count. Using a `CountDownLatch`, the worker tasks who countdown are *Providers*, and the master task who awaits is the *Accessor*. Ideally, there should be permissions that allow the worker to only call `increment` when the latch is closed, and assert the master can only access through `retrieve` method once it is open. Unfortunality, cannot work at the method level and the solution involves breaking the Data object up into three parts:

```
        public class Data {
            final AtomicInteger count = new AtomicInteger();

            public Data() {
                ICP.setPermission(this, Permissions.
    getPermanentlyThreadSafe());
            }

        }

        public class Accessor {
            final Data data;

            public Accessor(Data data) {
                this.data = data;
            }

            public int retrieve() {
                return data.count.get();
```

[6]Previous ICP revision did method level. (confirm???)

```
        }
    }

    public class Provider {
        final Data data;

        public Provider(Data data) {
            this.data = data;
        }

        public void increment() {
            this.data.count.incrementAndGet();
        }
    }

    // ... Somewhere after initialization and setting up synchronizer
    ICP.setPermission(accessor, <permission-associated-with-master>);
    ICP.setPermission(provider, <permission-associated-with-worker>);
```

The problem is `Data` object is thread safe as both the accessor and provider need it. This is solved with task-based permission.

7 One-time latch

Count down latches of 1. `isOpen()` opens latch, `await()` waits for it to be open. Unlike ICP CountDownLatch, open may be called multiple times because in the case of multiple workers, only one should open the task, but any one may nondeterministically be the opener. This includes having multiple openers in registration. By using an AtomicInteger to keep track of remaining finishes tasks, it is possible to only call `open` once, but all have to be registered to call open.

```
@Override
public void jobCompleted(WordResults job) {
    completeLatch.registerOpener(); // Any task may be an opener
    shared.addResult(job);
    int left = jobsLeft.decrementAndGet();
    if (left == 0) {
        // But only one task *opens* it
        completeLatch.open();
    }
}
```

Listing 3: Multiple openers

With only one opener, Listings 3 [7] would be impossible.

[7] See `applications.forkjoin.synchronizers.OneTimeLatchRegistration`

7.0.1 Without registration

Only has *IsOpen* permission associated with synchronizer. With now registration, is is possible for the worker task who called `open()` to continue accessing the shared data.[8] See `applications.latches.OneWorkerOneShared` for example.

The disadvantage is the worker may still work on the shared object after calling `open()`. This is fixed with the one-time latch using registration.

7.0.2 Registration

Similar to the non-registration latch, but tasks must explicitly tell the synchronizer they are either a *opener* or *waiter*. This is done by calling `registerOpener()` and `registerWaiter()` methods.

With registration, the *Three Object Idiom* can be replaced with a single Data Object and single permission `getPermission()`. This permission first checks the current task and has multiple cases depending on opener or waiter.

See `applications.latches.OneWorkerOneRegistration` for example.

8 CountDownLatch

The General CountDownLatch synchronizer with a positive count and operation registration. There is no non-registration version because of the problems with 1-time latch (non-registration).[9]

Tasks must register as a *CountDowner* or *Waiter* using the methods `registerCountDowner()` and `registerWaiter()`. It is a violation if a *CountDowner* calls `countdown` more than once or the latch is open.

See `applications.latches.SimpleCountDown` for example.

9 Futures

Same as j.u.c's futures. Instead of building a synchronizer for Futures, it is up to the user to explicitly set the future's result permission.

Trade-offs

- Doesn't require list of if branches checking the current permission[10] and seeing if it needs to be set to *Transfer* before future returns.
- By forcing the user to set permission on result, they know by looking at the code how to use the future's result.

[8]IsOpen permission checks if latch is open, but does not know whether the current task is a worker or master.

[9] Just for fun, a non-registration CountDownLatch can be implemented, but only to prove registration fixes the problems which workers may still access after calling countdown.

[10]Not possible in current system

- There is no good way of warning the user they should set a permission before "exporting" result. This can lead to confusing IntentErrors.
- Allows users to keep using j.u.c's FutureTasks.

9.0.1 Transfer Permission

`applications.futures.SimpleFuture` example submits a FutureTask which creates a Result object, sets that object's permission to Transfer, and returns it. The task who submitted object calls `future.get()` and now has accesses to Result by transfer.

```
    Future<Result> future = ForkJoinPool.commonPool().submit(() -> {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        Result result = new Result(42);
        ICP.setPermission(result, Permissions.getTransferPermission());
        return result;
    });

    assert future.get().value == 42;
```

9.0.2 Callable interface

The Callable interface used when submitted jobs that return results require Callables to be Tasks. The `core.CallableTask` does not extend Task, but encapsulates one because the callable should not be join-able and the "has-a" relationship on Runnable doesn't exist. The CallableTask implements Callable interface, but the factory methods export the interface only.[11]

The implementation uses a Task-runnable and AtomicReference as a "box" to pass from runnable to result of call method. Creation through factory methods supporting thread-safe and private callables interfaces. When passing result to and from the "box", no permissions are set. It is up to the user to set explicit permissions in their call method.

9.0.3 Problems

User does not set correct permission

The transfer permission allows any task to acquire rights to the object by accessing. If multiple tasks try to access it by calling `future.get()`, one will win, and the others will fail. This is a error on the user's part since they are allowing a *private* object to be shared among multiple tasks. This is asserted by the transfer permission.

Either the result is transferable and private to only one task, or made immutable or thread-safe for multiple tasks.

[11]Should it export CallableTask? User could cast it unless made package-private. There is no other operations besides `call`

Uses normal runnable instead of Task
See applications.futures.Bad1. User may forget to wrap their Runnable in a Task ready runnable (`Task.fromThreadSafeRunnable()`).

```
        Future f = ForkJoinPool.commonPool().submit(() -> {
            try {
                latch.registerOpener(); // Fails here as Task.currentTask
    is used
                data = 42;
                latch.open();
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
```

```
icp.core.IntentError: thread 'ForkJoinPool.commonPool-worker-1' is not a task
```

Uses normal callable instead of CallableTask
See `applications.futures.Bad2`. User may forget to wrap their Callable in a CallableTask ready callable.

```
        Future<Data> f = ForkJoinPool.commonPool().submit(() -> {
          try {
            return new Data(42); // Fails here in <init> of Data object
          } catch (Exception e) {
            e.printStackTrace();
          }

          throw new AssertionError("Should not reach");
        });
```

```
icp.core.IntentError: thread 'ForkJoinPool.commonPool-worker-1' is not a task
```

9.1 Lazy evaluation

In Scala, fields can be marked as `lazy` and are only computed when accessed for the first time and is thread-safe. Lazy evaluation can be implemented in Java by using Futures. See `applications.futures.Lazy`