ICP Notes

Kyle Kroboth

November 5, 2017

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

Contents

1 General

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 Calls to Lambdas are thread-safe

`checkCall` permission checks are not inserted before lambdas. Therefor lambdas are thread-safe. The body of lambdas are edited; field writes and reads.

1.3 Synchronizers

Practices used when creating library synchronizers.

Final classes All synchronizers should be **`final`** or not extendable. For example, `OneTimeLatchRegistration` has similar code in `OneTimeLatch`, but should not be a "is-a" relationship.

Registering methods As task-based registration is becoming more relevant, code for registering multiple methods is repeated, and multiple Boolean TaskLocal's are used. There should be in the future a abstract class which handles method registration or use a single TaskLocal that keeps track of multiple method calls.

TODO: A utility or common code for checking disjoint registrations is required.

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/rebellabs/l
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.

Possible solution 1

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.

Locking scenario With monitor locks it is impossible[1] to enter a monitor lock and exit in another method. Using the j.u.c Locks could allow one Task acquiring the lock, but having another task release it. This is only possible when the Thread runs two different tasks. This is avoided in `SimpleReentrantLock` by setting Task owner.

Possible solution 2

This was talked about on 10/30/17 and needs clarification, but instead of modifying the bytecode to replace monitors with ICP locks the `Task.holdsLock()` could be used. This method can return false positives. [2].

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[3], 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[4] 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.

TODO: Explain scenarios were @External annotations fail. Not that important right now.

[1] Possible through bytecode manipulation
[2] No notes why
[3] Right now there is no way to exclude entire or include specific packages
[4] See PermissionSupport.addPermissionField()

4 Synchronizers with Registrations

Registration refers to tasks saying *I'm going to perform this operation* before calling it. For example, for the case of 1-time latches, a Task will inform the synchronizer it is an *opener* before calling `oneTimeLatch.open()`. This allows a disjoint set of opener and waiter tasks.

As a side-effect of registration, Task-Based Permissions are possible. One-time latch permissions may include *IsOpenPermission* and *IsClosedPermission*. To use these permissions, three objects must be used: *Data*[5], *Accessor*, and *Provider*. This makes coding awkward. See Three Object Idiom for more information.

4.1 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.

The downside is there can only be one permission for a single 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.

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());` [6] Too complicated to implement and gets worse after composing more than one permission.

Solution: Registration

By using Boolean TaskLocal's in the synchronizer, it is possible to have one permission [7] 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());

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

[5] Usually thread-safe

[6] 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.

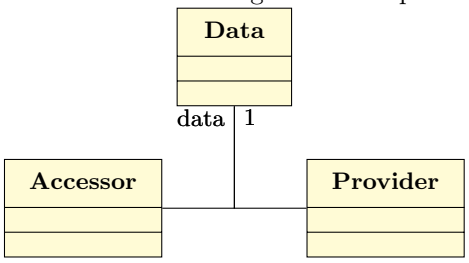
[7] *The Permission* to rule them all. For example, 1-time latch with registration has only one permission for both IsClosed and IsOpen checks.

```
        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.

5 Three Object Idiom

Permissions are tied at the Object level and not method.[8] 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, this cannot work at the method level and the solution involves breaking the Data 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();
    }
}
```

[8] 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.

6 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 [9] would be impossible.

6.1 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.[10] 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.

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

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

6.2 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.

7 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).[11]

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.

8 Futures

Multiple implementations of Futures (FutureTask in j.u.c) are in consideration as there is no general Future that can catch all common User errors, handle permissions, and easy for the user to use.

TODO: Note, there is some confusing of the work *task*. Must clarify when refering to ICP Task, FutureTask, or the common word task used for runnable/callable passed in executor.

8.1 Ideal Properties

Note, not all properties may be used at once and some contradict each other.

Property 1. Keep using j.u.c FutureTask without custom ICP implementation

Property 2. Implicit Task creation when passed a Runnable or Callable interface

Property 3. Explicit Task creation using factory methods defined in Task and CallableTask.

Property 4. Useful intent errors when using Futures incorrectly. Requires custom synchronizer.

Property 5. Explicit transfer of private object created in future before returned.

Property 6. Implicit transfer of project object using multiple if-statements to check whether permission may be changed, and if, change it to transfer.

8.2 Uses of Futures

Use 1. Task who creates the Future is the only task that may retrieve the result (Private to task)

Use 2. Use Future as a "joinable"[12] task. A custom synchronizer could have a `join()`[13] method that returns void. Could it be the case where multiple tasks call join(), but only one calls get()? Impossible in current ICP version as permissions are object based.

Use 3. A Future that allows multiple tasks retrieving a thread-safe or immutable object from `get()`.[14]

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

[12] Wait for runnable to finish.

[13] Make `ICPFuture extend Task`

[14] Emulate Scala's Lazy val. See `applications.futures.Lazy`

8.3 Common code used in examples and implementations

```
// Return object from Callable
class Result {
    // Neither final or volatile to express memory semantics of Futures
    int value;

    Result(int value) {
        this.value = value;
    }
}

class ImmutableResult {
    final int value;

    ImmutableResult(int value) {
        this.value = value;
        ICP.setPermission(this, Permissions.getFrozenPermission());
    }
}
```

8.4 Implementation A: CallableTask Wrapper

Create a `CallableTask` class which implements `Callable` and pass the wrapped user callable to j.u.c Executors. No special treatment of transfering the callable's return object's permission is done. This is left up to the user.

How to Use

```
Future<Result> future = executorService.submit(CallableTask.ofThreadSafe(() ->
     {
    Result result = new Result(42);
    ICP.setPermission(result, Permissions.getTransferPermission());
    return result;
}));
assert future.get().value == 42;
```

Listing 4: Wraping callable in CallableTask

```
executorService.submit(Task.ofThreadSafe(() -> {
    Result result = new ImmutableResult(42);
    sharedCollection.add(result);
}));
```

Listing 5: Wrapping runnable in Task

Both examples rely on wrapping the Runnable and Callable interfaces in ICP compliant Tasks. The benefit is the user may still use their own Executors and the wrapping of interfaces is not hidden.

What problems may occur if used incorrectly

Downsides are not being able to assert ICPTasks or ICPCallables sent into the Executor. If a user forgets to wrap their interface, undefined behavior is possible.

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.

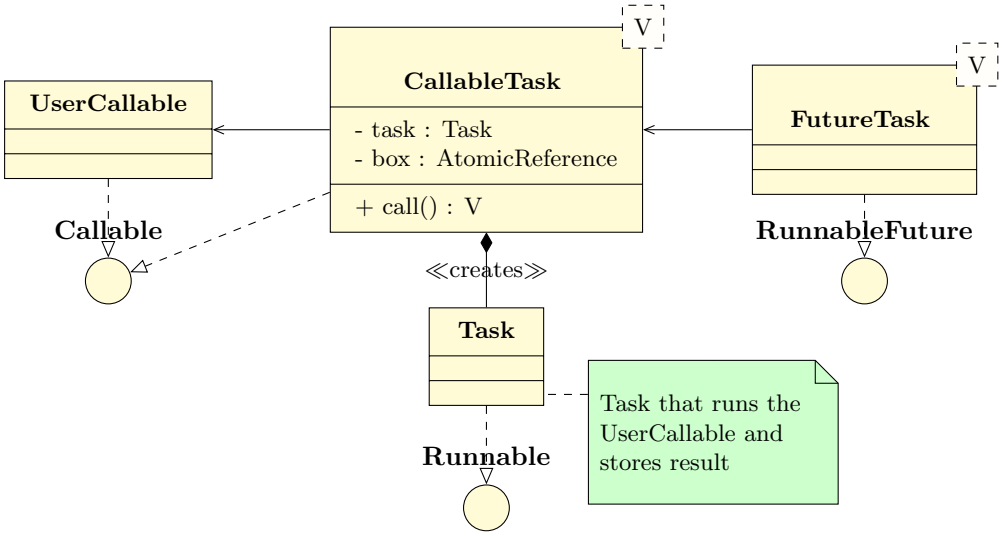
Either the data is transferable and private to only one task, or made immutable or thread-safe for multiple tasks.

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

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

Forcing a Callable to be Task



The Callable interface used for submitted jobs in the Executor must be run by a ICPTask. ICPTasks are not of Callable type and therefore a custom ICPCallable middle-man must handle the result.

The ICPCallable does not extend Task, but encapsulates it 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.[15]

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

It should be noted that FutureTask when passed a Callable *does not* wrap it in another layer to handle returning the result. This is only done when passed a Runnable.

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

Trade-offs

- Doesn't require list of if branches checking the current permission[16] and seeing if it needs to be set to *Transfer* before future returns.
- By forcing the user to set permission on data, they know by looking at the code how to use the future's data.
- There is no good way of warning the user they should set a permission before "exporting" data. This can lead to confusing IntentErrors.
- Allows users to keep using j.u.c's FutureTasks.

Summary

Allows user to keep using j.u.c Executor and FutureTask (Property 1). Explicit Task creation and permission setting (Properties 3 and 5).

If future's result is used in multiple tasks and result is private, the IntentError would not be caught until task touched result (Issues with Use case 1).

8.5 Implementation B: Wrapping RunnableFuture in Task

The Java Executor interface has one method `execute(Runnable)` and all the other task execution methods (callables, runnables, runnables with result) delegate to it. The original problem was not having the user's callables run in a Task environment. A Task wraps a j.u.c FutureTask.

How to use

```
RunnableFuture<Result> future = new FutureTask<>(() -> {
  Result result = new Result(42);
  ICP.setPermission(result, Permissions.getTransferPermission());
  return result;
});

executorService.execute(Task.ofThreadSafe(future));
assert future.get().getValue() == 42;
```

Listing 6: Wrap RunnableFuture in Task

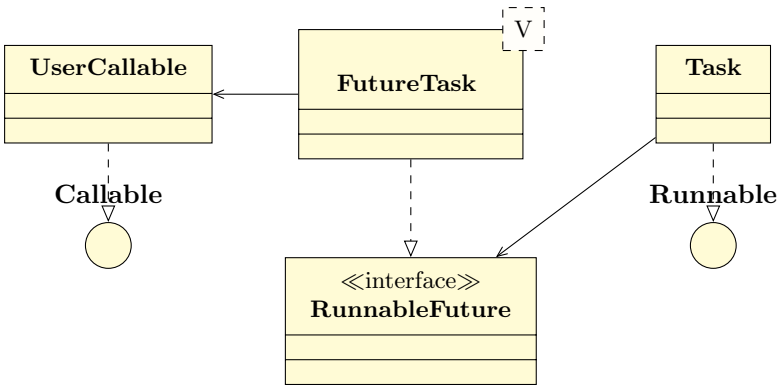
By still using j.u.c Executors, the user must not use the submit methods which take a Callable, but build a FutureTask instead. For normal runnables, `executorService.execute(Task.ofThreadSafe(userRunnable))` is enough.

What problems may occur if used incorrectly

All of the same problems "Implementation A: CallableTask Wrapper" may happen.

[16] Not possible in current system

Wrapping classes



Only one level of wrapping of RunnableFuture inside a Task is required.

Trade-offs

- No checking of permissions as the user must set it.
- Know how the returning object must be used (Explicit permission set)
- Still no good way of warning users if incorrect Runnables or Callables are sent in executors
- `execute(Runnable)` in Executor interface is the only method that may be used.

Summary

Has the same properties and uses as "Implementation A: CallableTask Wrapper". This implementation is the simplest from ICP perspective, but most troubling to the user as they may forget to wrap FutureTasks.

8.6 Implementation C: ICPFutureTask

Implement a ICPFutureTask that allows permission to be set on it and handles wrapping Callables and Runnables. Having permissions on a Future allows futures to be private to the task who created it. Once again, this allows using the same executors in j.u.c, but the user *must* use the `execute(Runnable)` passing the RunnableFuture.

Actual Implementation

The actual implementation of `core.FutureTask` extends `Task`. This allows not only using Task methods join(), but the passed in callable or runnable does not have to be wrapped. Even if the user does happen to wrap a runnable in a Task before sending it future, it still works. Same as wrapping the entire RunnableFuture in Task before sending to execute method.

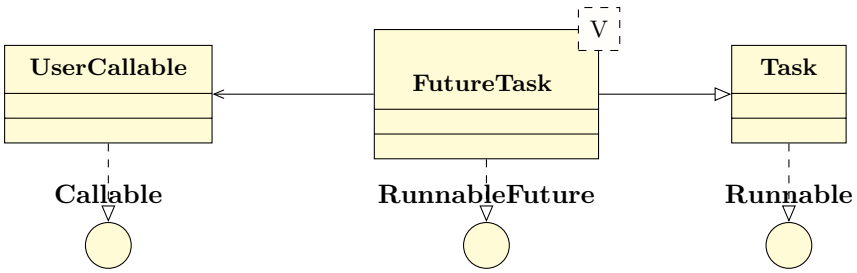
The constructors of Task allow a null task and atomic running boolean set to null internally for InitTask only [17] or a constructor that takes a runnable. Since our FutureTask delegates to a j.u.c FutureTask, it must be created before passing the runnableRunnableFuture in super call. For now, a static dummy runnable is always passed in and never used.

The passed in runnable is never used due to another method on Task added: `doRun()`. This package-private method is called in `Task.run` and allows overridable task execution. By default, it runs the runnable passed in Task constructor.

Another downside is having to call `Permissions.checkCall(this);` in the first line of all methods. This FutureTask resides in the core package which is ignored.

[17] Only for InitTask?

Wrapping classes



No extra wrapping outside of FutureTask wrapping a Callable inteface, which is required.

Variant checking passed in Runnables
It is still valid to wrap a runnable in a Task before sending it off to the FutureTask, but it only adds an extra unneeded layer. Assertions can be added to avoid this in the constructors. It would still be possible to wrap the FutureTask in a Task before sending it off to an executor.

Trade-offs

- Control over callable or runnable passed in through ICPFutureTask
- Setting permission on ICPFutureTask. Multiple calls to get now fail if private.
- Still explicit result permission required by user.
- `execute(Runnable)` in Executor interface is only method that may be used.

Summary

An improvement over past implementations as there are no extra unnecessary wrapping of Callable's, and allows permissions to be set on ICPFutureTask. Properties 2, 4, 5 are satisfied. All Use cases can be used without any trouble.

8.7 Implementation D: ICPExecutor

Until now, executors are still ones from j.u.c without a custom ICP executor. This implementation adds a new interface `ICPExecutorService` that extends `ExecutorService`. The interface methods are overridden to return ICPFutureTask instead of Future. No new methods are added.

Methods which take a Callable are sent into the ICPFutureTask from "**??**" and the same code is used. Methods taking a Runnable are either wrapped in a Task or sent as is.

```
delegate.execute(command instanceof Task
? (Task) command
: Task.ofThreadSafe(command));
```

Listing 7: execute(Runnable)

Tasks can still be passed in this method and wrapped if a non-task runnable runs a Task inside its run method. It is up to the user to avoid this, but has no effect on the system.

Variants of execute(Runnable)
Instead of wrapping a non-task runnable, an IntentError can thrown to assert only Tasks are sent in. Another option is adding a new execute method that only takes Tasks and the other execute(Runnable) method always throws an IntentError.

Automatically wrapping a non-task runnable is a cleaner approach and allows users to use the `ExecutorService` interface.

InvokeAll and InvokeAny
These methods are not implemented and go directly to the delegated Executor. May not be trivial implementations[18]

One idea is to take the `Collectoin<? extends Callable>` parameter in `invokeAll` and `invokeAny` methods and wrap them in a CallableTask from "Implementation B: Wrapping RunnableFuture in Task Collection. This is not implemented and awaiting discussion.

How to use

```
// Wrap j.u.c executor in ICP executor.
ICPExecutorService executor = ICPExecutors.newICPExecutorService(
    Executors.newCachedThreadPool());

// Callable
FutureTask<Result> future = executor.submit(() -> {
  Result result = new Result(42);
  ICP.setPermission(result, Permissions.getTransferPermission());
  return result;
});
assert future.get().getValue() == 42;

// Runnable
future = executor.submit(() -> {
  // ICP permission check in runnable
  assert immutableResult.getValue() == 43;
}, new Result(42));
assert future.get().getValue() == 42;

// Runnable and using Join Permission instead of get.
Result box = new Result(0);
FutureTask<?> futureRunnable = new FutureTask<>(() -> {
  assert immutableResult.getValue() == 43; // permission check
  box.setValue(43);
}, null);
ICP.setPermission(box, futureRunnable.getJoinPermission());
executor.execute(futureRunnable);
futureRunnable.join();
assert box.getValue() == 43;
```

8.8 Using all implemetations at once

There are four implementations of Futures and all *could* be kept in the system and left up to the user to decide. However, the goal was to wrap code whether a runnable, callable, or future in a task and wrapping multiple tasks inside of each other is still valid.

Consider wrapping a callable inside a ICPCallable, then passing it in the ICPFutureTask, and finally wrapping the future in a Task to send to the ICPExecutor.

```
ICPExecutorService icpexecutor = ICPExecutors.newICPExecutorService(Executors.
    newCachedThreadPool(
Utils.logExceptionThreadFactory()
));
future = new FutureTask<>(CallableTask.ofThreadSafe(() -> {
    Result result = new Result(42);
    ICP.setPermission(result, Permissions.getTransferPermission());
    return result;
}));
```

[18] http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/concurrent/ExecutorService.java

```
icpexecutor.execute(Task.ofThreadSafe(future));
assert future.get().getValue() == 42;
icpexecutor.shutdown();
```

Listing 8: Multiple implemetations simultaneously

8.9 Solution

The ICPExecutor and ICPFutureTask implementations are best suited as they allow implicit conversion to Tasks and don't introduce any extra wrapping of Callables to Tasks or Futures to Tasks. Since ICPExecutorService extends ExecutorService, once the user wraps juc Executor, it can still use the inteface without updating their code.

InvokeAll and InvokeAny methods are not supported until further discussion. The CallableTask implementation can be used to wrap the collection of Callable's before delegated to underlying juc Executor. However, this CallableTask should be package-private and the user will have no idea, as long as it works. Avoiding the extra wrapping is not possible.

8.10 Applications

Scenarios using Futures. TODO: Write more applications!

8.10.1 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`

9 Joining Tasks

A common practice of fork-join implementations involves spawning multiple worker threads, join on all of them, and access the shared data. ICP works with Tasks, and not Threads which means the following code will not work:

```
// Fork
for(int i = 0; i < 10; i++) {
    threads[i] = new Thread(() -> {
        data[i] = new Result(i);
    });
}

// Join
for(int i = 0; i < 10; i++) {
    threads[i].join();
    data[i].getResult(); // worker done with data
}
```

See `applications.joins.Bad1` for complete example.

There is a happens-before relationship of all the actions in each worker and then calling join() on same worker thread. However, no JMM checks are implemented in ICP so there needs to be a Join permission on result objects.

This cannot be checked in the current system and the user must use permissions on objects that may be accessed after a given Task has joined[19]. Every task has a `join` and `getJoinPermission`

[19]Better name is completed, finished, calling await()

methods. Task-registration is used implicitly here as the current task running has permission access the object until the task has finished. The master task calls `join` which registers itself.

```
Task task = Task.ofThreadSafe(() -> {
  try {
    Thread.sleep(10);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }

  data.counter = 1;
});
// set join permission on data
ICP.setPermission(data, task.getJoinPermission());

// throw away access to thread -- use task.join()
new Thread(task).start();

// join and get access to data
task.join();
assert data.counter == 1;
```

Listing 9: Example of Task.join()

blueTODO: Mention failed TaskThread, TaskThreadGroup, and force setting of permissions which violates Hatcher's principle. Not used anymore.

10 Semaphores

Permissions
Each semaphore keeps a TaskLocal count of acquired permits for each task. The permission will assert that that task has acquired at least one permit.[20].

10.1 Disjoint Semaphore

Semaphore were acquriers and releasers tasks are disjoint. A suitable scenario is a `Bounded Buffer`.

10.2 Release Semaphore

Semaphore that requires the same Task who acquires N permits to release N permits.

Possible Implementation How restrictive should *the task who acquired N permits must release them* be? Once a calls `acquire(N)` or `acquire()` it must the same amount of N permits before calling another acquire.

[20]Should at least N permits be a different permission or synchronizer?

Glossary

ICPCallable Task compliant callable inteface.. 9

ICPTask ICP Task class in `icp.core.Task`. ICP prefix is used to not confuse reader on common word *Task* which may refer to FutureTasks, or a computational task.. 9

j.u.c The `java.util.concurrent` package. 3, 7, 8, 10–12

User The person who uses ICP library to express their intentions on how objects are shared. 3–5, 7