Intentional Concurrent Programming

Kyle Kroboth

Advisors: Michel Charpentier and Philip Hatcher

Department of Computer Science
University of New Hampshire

Introduction

Multithreading improves the speedup, responsiveness, and throughput of an application, however accessing shared memory from multiple threads can result in race conditions making concurrent programming very difficult.

Solution:

- Give mechanism to express intents on shared objects between threads
- Develop intent-aware synchronizers with permissions
- ICP System raises exceptions when intents violated

Implementation In Java

Edit bytecode of classes loaded and insert checks on field accesses and method calls. Every object is associated with a permission field that holds its current intent.

- Uses Javassist bytecode manipulation library
- Extend java.util.concurrent standard synchronizers
- Wrap collections with proxies containing the intent

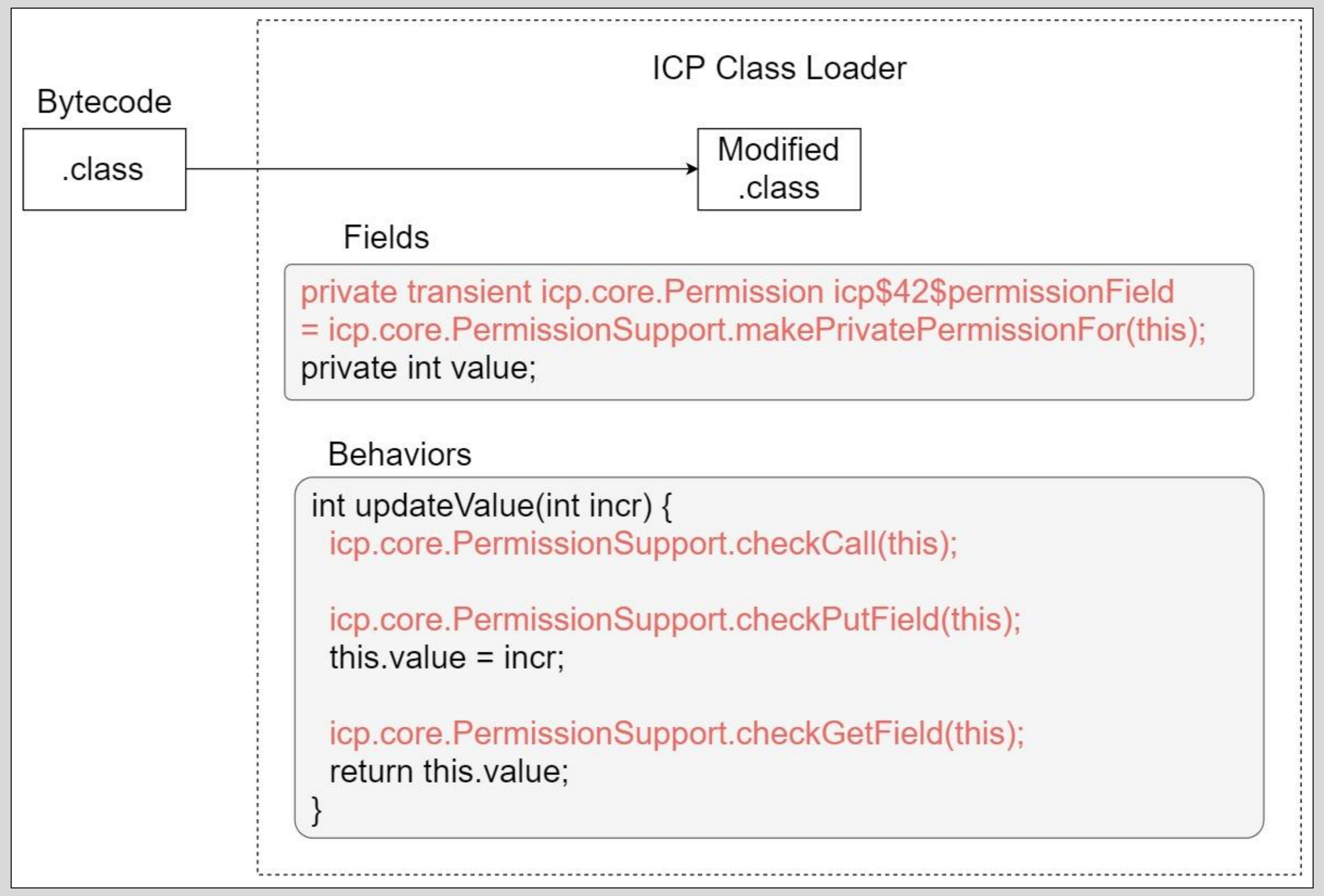

Figure 1: Bytecode manipulation

Permissions

Provide users with a mechanism to create intents and to associate them with target objects in order to check method calls and field accesses.

- One permission per object
- Initial permission: private to task that creates the object
- Built-in and user defined permissions
- Synchronizers provide their own permissions
- Some permissions are non-resettable, i.e., ThreadSafe

Provided Permissions

- Private (Initially)
- ThreadSafe (Non-resettable)
- Frozen (Immutable)
- Transfer (Give ownership)
- SameAs (Inherit parent)
- HoldsLock (Intrinsic lock)
- Join (Task joining)
- Latch (CountDownLatch)
- IsOpen/IsClosed (OneTimeLatch)
- Locked (ReentrantLock)
- AwaitTermination (Executor)
- Compound (Multiple)

Synchronizers

Intent-aware synchronizers extend Java's concurrent package with permissions.

- Export one or more permissions
- Checks the correct use of a synchronizer
- Registration allows synchronizer to behave differently for different Tasks

```
Data data = new Data();
SimpleReentrantLock lock = new SimpleReentrantLock();
ICP.setPermission(data, lock.getLockedPermission());
```

Guard data with reentrant lock

```
lock.lock();
try {
  data.updateValue(42);
} finally {
  lock.unlock();
}
```

Satisfies shared object intent

Intent Error!

```
data.updateValue(42); // No locking!
```

checkCall throws IntentError exception (see figure 1)

Figure 2: Correct vs incorrect synchronized code

Application

Develop a real-world server application using ICP and its synchronizers. Website displays travel information using multiple APIs including weather, restaurants, and events.

- Multithreaded HTTP 1.1 server that handles requests in parallel
- Each request also uses parallel subtasks and aggregates APIs into one payload based on the users selected location
- Utilized intent-aware CountDownLatch, Semaphore, Executor, and wrapped intent collections

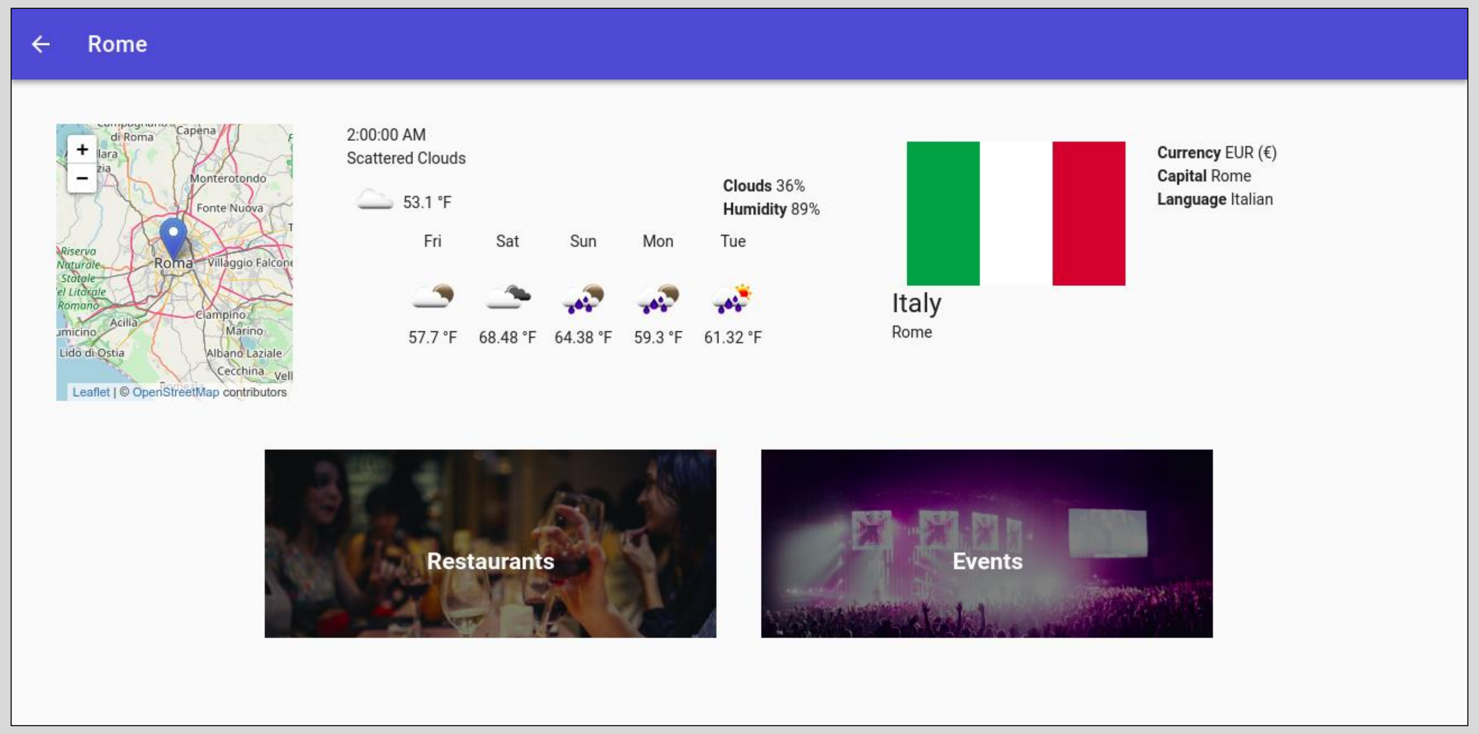

Figure 3: Screenshot of "Travel Info" application

Express intents on shared payload

```
CountDownLatch latch = new CountDownLatch(apiRequests.size()); // Intent-aware
ICPWrapper<JsonObject> payload = ICPWrapper.of(new JsonObject()); // Wrap 3rd party class
ICP.setCompoundPermission(payload, Permissions.getHoldsLockPermission(payload),
   latch.getPermission());
```

Latch registration intent

Synchronize payload intent

```
latch.registerWaiter();
```

Fork API calls using Tasks

```
apiRequests.forEach((name, req) -> { // Fork api requests
  ICP.setPermission(req, latch.getPermission());
  requestExecutor.execute(Task.ofThreadSafe(() -> {
    try {
      latch.registerCountDowner();
      JsonElement content = req.execute(httpHelper).getExtractedJson()
        .orElseThrow(() -> new IOException("Request content missing"));
      synchronized (payload) {
        payload.target().add(name, content);
      }
    } catch (IOException e) {
      logger.log(Level.SEVERE, e.getMessage(), e);
    } finally {
      latch.countDown();
    }
  }));
});
```

Checks permissions on shared payload object

Await tasks and return response

```
// Wait for requests to complete
try { latch.await(); } catch (InterruptedException e) { throw new HttpException(404, e); }

synchronized (payload) {
  JsonObject json = payload.target(); // Access shared json
  return Response.create(200).body(GSON.toJson(json));
}
```

Checks permissions on shared payload object

Figure 4: Intent code in application fetching API requests and building aggregated response

Results

The ICP runtime system provided useful feedback on incorrectly synchronized objects and performance was good at catching concurrent bugs early and ensuring thread safe code.

- Explicit permissions in code gave insight on how objects were shared
- Concurrency intents were fully expressed within ICP library
- The permission interface was able to fulfill multiple synchronization strategies

Future

Advanced patterns of concurrent programming will be evaluated in a future iteration. Atomicity bugs such as non-locking check-then-act, and subtle memory consistency errors are not handled in the current ICP system.