ECE 587 – Hardware/Software Co-Design
Lecture 09 Threads and Actors

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

February 9, 2026

Reading Assignment

- This lecture: Threads and Actors
- Next lectures: Memory Ordering

Outline

Processes and Threads

- A two-level scheme used by most modern OS to support multitasking and multiprocessing.
  - Hide details of processors/cores.
  - Runtime environments like JVM usually directly utilize such scheme for efficiency.
- An OS thread corresponds to a process as a sequential program.
  - Each OS thread belong to an OS process.
- An OS process provides a shared environment for its threads.
  - Shared virtual memory space.
  - System resources like file handles.
- Multi-threading: the OS process starts with its main thread.
  - This main thread may create more threads within the same OS process.
  - The process may or may not exit if the main thread terminates.

Java Threads

```
public class Test {
  public static void main(String[] args) {
    KVStorage storage = new KVStorage();
    Thread p0 = new Thread(() -> {
      storage.put("a", "hello");
    });
    Thread p1 = new Thread(() -> {
      storage.put("b", "world");
    });
    Thread c0 = new Thread(() -> {
      System.out.println(storage.get("a"));
    });
    Thread c1 = new Thread(() -> {
      System.out.println(storage.get("b"));
    });
    p0.start(); p1.start();
    c0.start(); c1.start();
  }
}
```

- It is actually not a good idea to create many OS threads as they are costly to create for most systems now.

Shared States

```
class KVStorage {

  public String get(String key) {
    return inner.get(key);
  }

  public String put(String key, String val) {
    return inner.put(key, val);
  }

  public String remove(String key) {
    return inner.remove(key);
  }

  private final HashMap<String, String> inner = new HashMap<>();
}
```

- Common OOP/OOD practice: encapsulate states in an object and provide CRUD operations.
- Race condition: not suitable for multi-threading.

Synchronization via Locks

```
class KVStorage {

  public synchronized String get(String key) {
    return inner.get(key);
  }

  public synchronized String put(String key, String val) {
    return inner.put(key, val);
  }

  public String remove(String key) {
    synchronized(this) {
      return inner.remove(key);
    }
  }

  private final HashMap<String, String> inner = new HashMap<>();
}
```

- Make use of intrinsic lock (built-in lock for each object).
- What's the output of our multi-threaded program?

Reason with Locks

- Regions protected by the same lock execute sequentially.
  - Lack of synchronization, i.e. control of ordering.
- Correctness
  - Non-deterministic: the actual ordering may differ from one execution to another. It is possible to see any two from `null`, `null`, `hello`, `world` in any order.
  - Deadlocks may exist.
- Performance
  - Contention may happen if multiple threads try to obtain the same lock at the same time.
  - The protected region may migrate across cores/processors, not cache friendly.
- Faulty behaviors
  - Troublesome if a thread holding the lock runs slow or dies.

Outline

Shared States as Actor

```
class KVStorage {
  public KVStorage() {
    new Thread(() -> {
      for (;;) try {actor(Q.take());} catch (Exception e) {}
    }).start();
  }
  private void actor(Token token) {
    if (token.type.equals("put"))
      inner.put(token.key, token.val);
    else if (token.type.equals("remove"))
      inner.remove(token.key);
  }
  private static class Token {
    final String type, key, val;
    Token(String t, String k, String v) {
      type = t; key = k; val = v;
    }
  }
  private final BlockingQueue<Token> Q = new LinkedBlockingQueue<>();
  ...
```

- As an actor receiving tokens from a single channel (queue).

Shared States as Actor (Cont.)

```
class KVStorage {
  ...
  public void put(String key, String val)
    throws InterruptedException {
    Q.put(new Token("put", key, val));
  }
  public void remove(String key)
    throws InterruptedException {
    Q.put(new Token("remove", key, null));
  }
  private final HashMap<String, String> inner = new HashMap<>();
}
```

- There should be another channel sending returned value back, in particular for get which we omit.
- No need to lock anything as the actor runs in the same thread – sequential execution guaranteed.
- Queue implementations may or may not contain a lock.

Thread Confinement

```
class KVStorage {
  public String get(String key) throws Exception {
    return exe.submit(() -> inner.get(key)).get();
  }
  public String put(String key, String val) throws Exception {
    return exe.submit(() -> inner.put(key, val)).get();
  }
  public String remove(String key) throws Exception {
    return exe.submit(() -> inner.remove(key)).get();
  }
  private final ExecutorService exe = Executors.newSingleThreadExecutor();
  private final HashMap<String, String> inner = new HashMap<>();
}
```

- Making use of Java executors can greatly simplify our code based on actors.

Future

```
class KVStorage {
  ...
  public Future<String> getAsync(String key) {
    return exe.submit(() -> inner.get(key));
  }
  ...
}
public class SomeClass {
  public void someFunction(String key) throws Exception {
    Future<String> f = storage.getAsync(key);
    ... // do a lot of computations not depending on val
    String val = f.get();
    ... // do a lot of computations depending on val
  }
}
```

- Use `Future` to enable asynchronous communication between actors.

Discussions: Correctness

- Lock-based and actor-based synchronizations for a single object are both easy to implement, leaving very small room for human mistakes.
- Both lead to sequential execution of protected regions, which greatly simplifies our reasonings.
  - Ordering of requests from the same threads follow the global ordering.
  - However, ordering of requests from different threads is non-deterministic.
- Deadlocks
  - Composing lock-based objects may lead to deadlock.
  - Actors are deadlock free if each of them uses its own thread/executor. However, many actors may share the same thread/executor in practice for efficiency, which may lead to subtle deadlocks.

Discussions: Performance

- Compared to lock where objects need to wait for the whole synchronized method to complete, actors have less contentions as one only need to wait for others to complete the queue operations.
- There exists queue implementations with very low synchronization overhead or support truely concurrent operations, making actors more attractive.

Discussions: Faulty Behaviors

- Failure of the actors attempting to write to a queue won't cause trouble for other actors to write to the same queue.
- Failure of the actor reading from a queue will lead to a full queue, which may be detected by an actor write to the same queue, and that actor may choose to react properly.
  - Is it possible to start a new actor to process the queue for fault recovery?
- For lock-based object, while it is possible to first detect that the thread holding the lock is dead and then reclaim the lock, it is generally impossible to recover from such fault as the shared state may already be inconsistent due to unfinished operations from the dead thread.
- In other words, from the aspect of fault resilience, it is much easier to deal with queues, which have very specific semantics, than lock-based objects that have arbitrary behaviors.