Lecture 9

- First mid-term: Wednesday, Feb 26
- No class on Monday, Feb 17
- Exam review on Wednesday, Feb 19 – Erlang Intro
- Video lecture on Monday, Feb 24
- Today:
  - Discuss Dekker algorithm
  - remainder of chapter 5

Dekker's Algorithm Properties – mutual exclusion

```
1.  boolean enter1 = false;
2.  boolean enter2 = false;
3.  int turn = 1;
4.  { while(true) { /* Thread 1 */
5.     enter1 = true;
6.     while (enter2) {
7.        if (turn==2) {
8.           enter1 = false;
9.           while (turn==2) skip;
10.          enter1 = true;
11.       }
12.    }
13.    /* critical section */
14.    enter1 = false; turn = 2;
15.    /* non-critical section */
16. }}
```

```
/* Thread 2 */
{ while(true) {
   enter2 = true;
   while (enter1) {
      if (turn==1) {
         enter2 = false;
         while (turn==1) skip;
         enter2 = true;
      }
   }
   /* critical section */
   enter2 = false; turn = 1;
   /* non-critical section */
}}
```

Dekker's Algorithm Properties – eventual entry

```
boolean enter1 = false;
boolean enter2 = false;
int turn = 1;
{ while(true) { /* Thread 1 */
   enter1 = true;
   while (enter2) {
      if (turn==2) {
         enter1 = false;
         while (turn==2) skip;
         enter1 = true;
      }
   }
   /* critical section */
   enter1 = false; turn = 2;
   /* non-critical section */
}}
```

```
/* Thread 2 */
{ while(true) {
   enter2 = true;
   while (enter1) {
      if (turn==1) {
         enter2 = false;
         while (turn==1) skip;
         enter2 = true;
      }
   }
   /* critical section */
   enter2 = false; turn = 1;
   /* non-critical section */
}}
```

Dekker's Algorithm Properties – non blocking

```
boolean enter1 = false;
boolean enter2 = false;
int turn = 1;
{ while(true) { /* Thread 1 */
   enter1 = true;
   while (enter2) {
      if (turn==2) {
         enter1 = false;
         while (turn==2) skip;
         enter1 = true;
      }
   }
   /* critical section */
   enter1 = false; turn = 2;
   /* non-critical section */
}}
```

```
/* Thread 2 */
{ while(true) {
   enter2 = true;
   while (enter1) {
      if (turn==1) {
         enter2 = false;
         while (turn==1) skip;
         enter2 = true;
      }
   }
   /* critical section */
   enter2 = false; turn = 1;
   /* non-critical section */
}}
```

Dekker's Algorithm Properties – no deadlock

```
boolean enter1 = false;
boolean enter2 = false;
int turn = 1;
{ while(true) { /* Thread 1 */
   enter1 = true;
   while (enter2) {
      if (turn==2) {
         enter1 = false;
         while (turn==2) skip;
         enter1 = true;
      }
   }
   /* critical section */
   enter1 = false; turn = 2;
   /* non-critical section */
}}
```

```
/* Thread 2 */
{ while(true) {
   enter2 = true;
   while (enter1) {
      if (turn==1) {
         enter2 = false;
         while (turn==1) skip;
         enter2 = true;
      }
   }
   /* critical section */
   enter2 = false; turn = 1;
   /* non-critical section */
}}
```

Cooperative Interruption

- Interruptions occur only when a thread is blocked
  - Interruption is delivered as a InterruptedException
- This is a nice model
- Thread is free to swallow an interrupted exception (not good practice)
- Contrast with C signals and signal handlers
  - Signal handler can start executing any time the signal is not blocked
  - This is not a nice model

Don’t swallow InterruptedException (Fig. 5.10)

- Methods that call things that can raise InterruptedException must either
  - Be declared as throw’ing InterruptedException, or
  - Catch InterruptedException
    - Clean up local state
    - Thread.currentThread.interrupt() // restore interrupt status if InterruptedException cannot be re-raised
  - Or both
    - Re-raising InterruptedException after cleaning up local state

Other fun synchronizers - Latches

- CountDownLatch – initial non-zero value
- Blocks threads calling latch.await() until latch value is 0; once 0 value never changes – "sticky" "monotonic property"
- latch.countDown() reduces the latch value
- Co-ordinated starting and ending points
  - Create N tasks that wait on a startingLatch (value 1)
  - startingLatch.countDown()
  - endingLatch.await() // endingLatch initial value N
  - Each of N threads calls endingLatch.countDown()

FutureTask

- Think of it as a thread body that computes a result value

- th = new Thread(future); // different constructor than Thread(runnable)
- future.get() returns the result of the execution – complicated by how to handle exceptions. Not very elegant to my mind

Semaphores

- Like latches, an integer value
- acquire(): if value is 0 wait, otherwise reduce the value by 1
- release(): increase the value by 1

Barriers

- **CyclicBarrier – allow N threads to repeatedly all gather at the barrier**
- **Think about rewriting the example used in Hwk1 to use latches and/or barriers in place of the homegrown barriers that I used.**

Results cache (5.6)

- Very interesting example
- Sometimes called a “memo” cache
- Illustrates implementation of caching – a common technique to increase performance
  - Idea: if it takes a long time to get an answer, save it for awhile in case you need it again
    - Memory caches, disk caches, results caches

Results Cache Basic Structure

- If (cache holds desired value) {
  return desired value
  } else {
  dv = compute-desired-value
  insert dv in cache
  return dv
  }

Stages in Cache Design

- **Cache for concurrent use is tricky**
  - **Obvious, use intrinsic lock around everything. Problem: lock held during long computation limits concurrency**
  - **Use ConcurrentHashMap – better concurrency but unsynchronized check-then-act with long "act" potentially allows unneeded work**
  - **Cache a FutureTask in the ConcurrentHashMap – still unsynchronized check-then-act but now the "act" is just inserting a FutureTask rather than the long expensive computation**
  - **Use putIfAbsent to eliminate this final race**

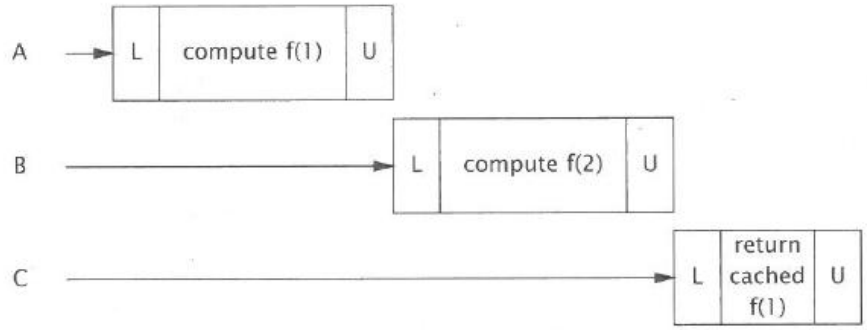


FIGURE 5.2. Poor concurrency of Memoizer1.

Stages in Cache Design

- **Cache for concurrent use is tricky**
  - **Obvious, use intrinsic lock around everything. Problem: lock held during long computation limits concurrency**
  - **Use ConcurrentHashMap – better concurrency but unsynchronized check-then-act with long "act" potentially allows unneeded work**
  - **Cache a FutureTask in the ConcurrentHashMap – still unsynchronized check-then-act but now the "act" is just inserting a FutureTask rather than the long expensive computation**
  - **Use putIfAbsent to eliminate this final race**

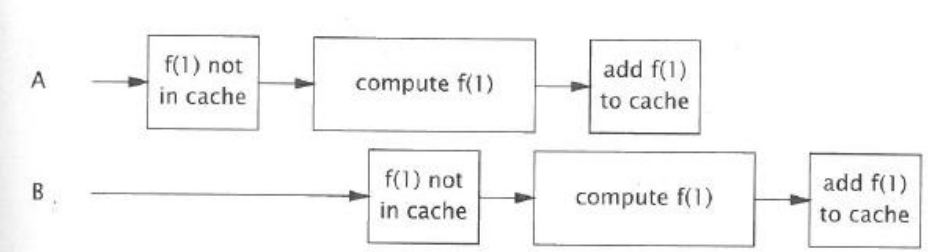


FIGURE 5.3. Two threads computing the same value when using `Memoizer2`.

Stages in Cache Design

- **Cache for concurrent use is tricky**
  - **Obvious, use intrinsic lock around everything. Problem: lock held during long computation limits concurrency**
  - **Use ConcurrentHashMap – better concurrency but unsynchronized check-then-act with long "act" potentially allows unneeded work**
  - **Cache a FutureTask in the ConcurrentHashMap – still unsynchronized check-then-act but now the "act" is just inserting a FutureTask rather than the long expensive computation**
  - **Use putIfAbsent to eliminate this final race**

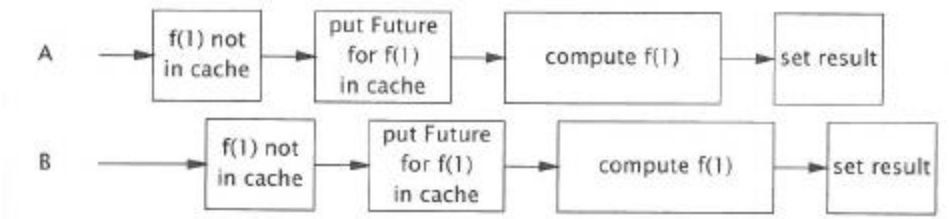


FIGURE 5.4. Unlucky timing that could cause Memoizer3 to calculate the same value twice.