

Concurrency WS 2010/2011

Termination Detection Barriers

Peter Thiemann

January 16, 2011

1 Termination Detection Barriers

Termination Detection Barriers

- Barriers up to now
 - Computation organized in phases
 - Barrier used to synchronize phase transition
- Different kind of barrier: termination detection
 - Thread pool: terminate when all threads have run out of work
 - More complicated than bare counting
 - Threads must reach consensus that all of them are inactive

Termination Detection Barrier Interface

```
1 public interface TDBarrier {  
2     void setActive (boolean state);  
3     boolean isTerminated ();  
4 }
```

- `setActive(true)`
is called **before** the thread starts looking for work
- `setActive(false)`
is called when the thread is definitively out of work
- `isTerminated()`
returns `true` when all threads are unemployed

Simple Termination Detection Barrier

```
1 public class SimpleTDBarrier implements TDBarrier {
2     AtomicInteger count;
3     int size;
4     public SimpleTDBarrier (int n) {
5         count = new AtomicInteger (n);
6         size = n;
7     }
8     public void setActive (boolean active) {
9         if (active)
10            count.getAndDecrement ();
11        else
12            count.getAndIncrement ();
13    }
14    public boolean isTerminated () {
15        return count.get () == size;
16    }
17 }
```

- Counter initialized to number of participating threads
- Transitions of each thread modifies the counter:
 - inactive \rightarrow active: decrements counter
 - active \rightarrow inactive: increments counter
- If all threads are inactive, then the counter reverts to the number of threads: termination!

Example Use: Work Stealing Executor Pool

```
1  public void run () {
2      int me = ThreadID.get();
3      tdBarrier.setActive (true);
4      Runnable task = queue[me].popBottom();
5      while (true) {
6          while (task != null) {
7              task.run();
8              task = queue[me].popBottom();
9          }
10         tdBarrier.setActive (false);
11         while (task == null) {
12             int victim = random.nextInt () % queue.length;
13             if (!queue[victim].isEmpty()) {
14                 tdBarrier.setActive (true);
15                 task = queue[victim].popTop();
16                 if (task == null)
17                     tdBarrier.setActive (false);
18             }
19             if (tdBarrier.isTerminated())
20                 return;
21         }
22     }
23 }
```

- A subtlety
 - Tests whether queue is empty (line 13) **before** declaring activity.
 - Otherwise, threads announce activity even if there is no chance of successfully stealing work.
- Proof obligations
 - Safety: if `isTerminated()` returns `true`, then the computation has indeed terminated.
 - No active task may declare itself inactive. (Other way round ok)
 - Liveness: if the computation terminates, then `isTerminated()` **eventually** returns `true`.
 - See above subtlety.