Language Support for Lightweight Transactions

(Tim Harris & Keir Fraser, OOPSLA'03)

Patrik Persson, Nov. 14, 2013

Java monitors are tricky!

```
public synchronized int get() {
   int result;
   while (items == 0) wait();
   items --;
   result = buffer[items];
   notifyAll();
   return result;
}
```

Back to the drawing board

- Conditional critical regions
(Tony Hoare, 1972)

- Atomic execution wrt. other atomic sections *accessing the same data*

- Where did the lock go?

```
public int get() {
   atomic (items != 0) {
      items --;
      return buffer[items];
   }
}
```

Software Transactional Memory (STM)

- Transactional memory:
  a memory model that checks ordering of memory accesses

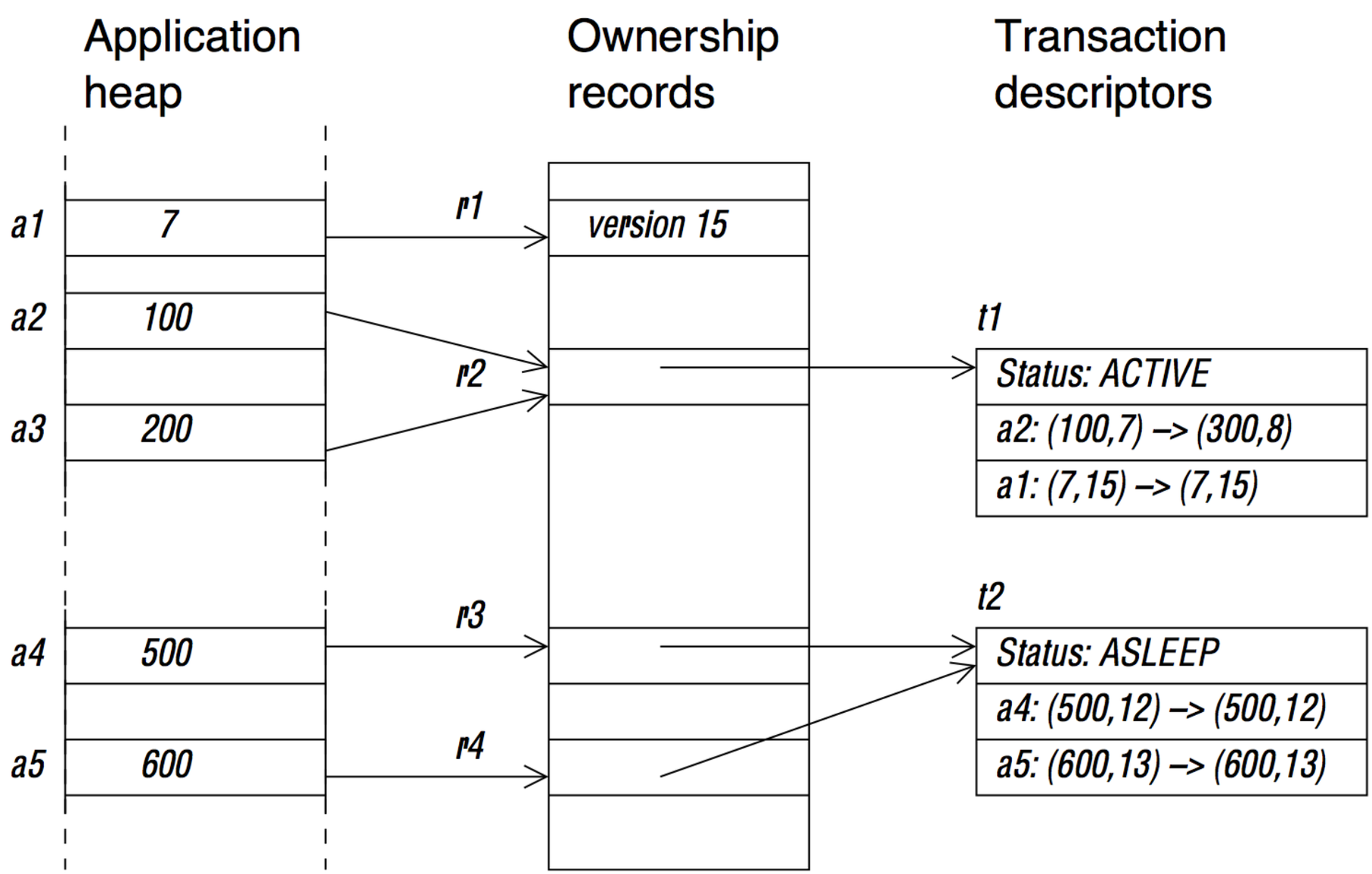
  - Optimistic access with recovery strategies,
    rather than conservative locking

  - Limited support in modern CPUs, e.g., Load-link & Store-conditional (MIPS, ARMv6, …)

- Software transactional memory:
  software-based approaches with similar semantics

  - Still relies on some CPU support, e.g., Compare-and-Swap

Tracking versions in memory



Deadlock, be gone!

```
atomic {
  synchronized(a) {
    synchronized(b) {
      ...
    }
  }
}
```

```
atomic {
  synchronized(b) {
    synchronized(a) {
      ...
    }
  }
}
```

Summary

- Declarative monitor-like concept,
  based on transactional memory

- They call it *non-blocking*, but it's really *non-locking*:
  blocking is possible (and intended) for boolean conditions

- Claim to avoid deadlock & priority inversion

- Fair performance,
  scales better than locking wrt. contention

Language Support for Lightweight Transactions

Exercises

STM exercises (1/2)

Consider the class *Fifo*. Assume multiple producers, multiple consumers.

1. There is (at least one) concurrency-related bug here. How can it be detected during testing?

2. Rewrite the *Fifo* class using *atomic*. How does this solution address the bug above?

```
class Fifo {
  public Fifo(int sz) { vals = new int[this.sz = sz]; }

  public synchronized int get()
    throws InterruptedException
  {
    if (r == w)  wait();
    int result = vals[r];
    r = (r + 1) % sz;
    notifyAll();
    return result;
  }

  public synchronized void put(int val)
    throws InterruptedException
  {
    if (r == ((w + 1) % sz))  wait();
    vals[w] = val;
    w = (w + 1) % sz;
    notifyAll();
  }

  private final int[] vals;
  private final int sz;
  private int r = 0;
  private int w = 0;        // empty when r == w
}
```

STM exercises (2/2)

Now consider the class *NumberSequence*. The method *someHeavyComputation()* is computationally intensive, and may have side effects.

3. This is thread-safe, but inefficient. Why?

4. If *atomic* is used, how might performance be affected? Explain the significance of transactions (STM) here.

```
class NumberSequence {
    …
  public synchronized void computeNext() {
    nbrs[pos++] = someHeavyComputation();
  }
  …
  public synchronized int size() {
    return pos;
  }
  …
  private int pos;
  private int nbrs[];
}
```