Worksheet #28 solution: use of tryLock()

Extend the transferFunds() method from Lecture 26 (shown below) to use j.u.c. locks with tryLock() instead of synchronized, and to return a boolean value --- true if it succeeds in obtaining both locks and performing the transfer, and false otherwise. Assume that each Account object contains a reference to a dedicated ReentrantLock object. Sketch your answer below using pseudocode. Can you create a deadlock with multiple calls to transferFunds() in parallel?

```java
1. public boolean transferFunds(Account from, Account to, int amount) {
2.     // Assume that each Account object has a lock field of
3.     // a type/class that implements java.util.concurrent.locks.Lock
4.     // Assume that no exception can be thrown in this code
5.     // Calls to this method can never lead to a deadlock
6.     if (! from.lock.trylock()) return false;
7.     if (! to.lock.trylock()) { from.lock.unlock(); return false; } 
8.     from.subtractFromBalance(amount); to.addToBalance(amount);
9.     from.lock.unlock(); to.lock.unlock();
10.    return true;
11. }
```
Design Patterns = formal discipline

• Christopher Alexander’s approach to (civil) architecture:
  – A design pattern “describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” Page x, A Pattern Language, Christopher Alexander, 1977

• A pattern language is an organized way of tackling an architectural problem using patterns

• The Design Patterns book turned object oriented design from an “art” to a systematic design discipline.
Example of OO Design Pattern: Visitor

1. class Employee {
2.   private int vacationDays; private String SSN;
3.   public void accept(Visitor v) { v.visit(this); }
4. ... 
5. }
6. abstract class Visitor {
7.   public abstract void visit(Employee emp);
8. }
9. class VacationVisitor extends Visitor {
10.  private int totalDays;
11.  public VacationVisitor() { total_days = 0; }
12.  public void visit(Employee emp) {
13.     totalDays += emp.getVacationDays();
14. }
15.  public int getTotalDays() { return totalDays; }
16.}
17. ... 
18. VacationVisitor v = new VacationVisitor();
19. emp1.accept(v); emp2.accept(v); ...
20. ... v.getTotalDays() ...
Patterns in Parallel Programming

- Can a pattern language/taxonomy providing guidance for the entire development process make parallel programming easier?
  - Need to identify basic patterns, along with refinements (usually for efficiency)
  - By relating HJ constructs to parallel programming patterns, you can apply HJ concepts to any parallel programming model you encounter in the future

- Algorithmic Patterns
  - Selection of task and data decompositions to solve a given problem in parallel
    - Task decomposition = identification of parallel steps
    - Data decomposition = partitioning of data into task-local vs. shared storage classes
  - Examples: Parallel Loops, Parallel Tasks, Reductions, Dataflow, Pipeline
## Selecting the Right Pattern
(adapted from page 9, Parallel Programming w/ Microsoft .Net)

<table>
<thead>
<tr>
<th>Application characteristics</th>
<th>Algorithmic pattern</th>
<th>Relevant HJ constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential loop with independent iterations</td>
<td>1) Parallel Loop</td>
<td>forall, forasync</td>
</tr>
<tr>
<td>Independent operations with well-defined control flow</td>
<td>2) Parallel Task</td>
<td>async, finish</td>
</tr>
<tr>
<td>Aggregating data from independent tasks/iterations</td>
<td>3) Parallel Aggregation (reductions)</td>
<td>finish accumulators</td>
</tr>
<tr>
<td>Ordering of steps based on data flow constraints</td>
<td>4) Futures</td>
<td>futures, data-driven tasks</td>
</tr>
<tr>
<td>Divide-and-conquer algorithms with recursive data structures</td>
<td>5) Dynamic Task Parallelism</td>
<td>async, finish</td>
</tr>
<tr>
<td>Repetitive operations on data streams</td>
<td>6) Pipelines</td>
<td>phasers, actors</td>
</tr>
</tbody>
</table>
How to select parallel constructs in general?

1. Think of how to decompose your program into tasks
   ⇒ async, future

2. Think of how to synchronize task creation and termination
   ⇒ finish, future-get, async-await

3. Think of where multiple tasks need to operate on shared data
   ⇒ Deterministic sharing: finish accumulators
   ⇒ Nondeterministic sharing: atomic variables, isolated, actors

4. Think of how to make your program more efficient
   ⇒ Recursive tasks: seq clause
   ⇒ Parallel loops: iteration grouping (chunking)
   ⇒ SPMD model: replace synchronizations in #2 by barriers/phasers
   ⇒ Isolated: use of atomic variables or object-based isolation

5. Think of when you need lower-level control beyond HJ-lib (should be rare)
   ⇒ Time-outs: Java threads and locks
   ⇒ Advanced locking: Java locks with tryLock()
Safety vs. Liveness

• In a concurrent setting, we need to specify both the safety and the liveness properties of an object

• Need a way to define
  — Safety: when an implementation is functionally correct (does not produce a wrong answer)
  — Liveness: the conditions under which it guarantees progress (completes execution successfully)

• Data race freedom is a desirable safety property for most parallel programs

• Linearizability is a desirable safety property for most concurrent objects
Liveness

- Liveness = a program’s ability to make progress in a timely manner
- Is termination a requirement for liveness?
  - But some applications are designed to be non-terminating
- Different levels of liveness guarantees (from weaker to stronger)
  1. Deadlock freedom
  2. Livelock freedom
  3. Starvation freedom
  4. Bounded wait
Terminating Parallel Program Executions

- A parallel program execution is terminating if all sequential tasks in the program terminate.
- Example of a nondeterministic data-race-free program with a nonterminating execution:

1. \( p.x = \text{false}; \)
2. \( \text{finish} \) {
3. \( \text{async} \) \( \text{// S1} \)
4. \( \text{boolean} \, b = \text{false}; \, \text{do} \, \{ \, \text{isolated} \, b = p.x; \, \} \, \text{while} \, (! \, b); \)
5. \}
6. \( \text{isolated} \, p.x = \text{true}; \) \( \text{// S2} \)
7. \} \) \( \text{// finish} \)

- Some executions of this program may be terminating, and some not.
- Cannot assume in general that statement S2 will ever get a chance to execute if async S1 is nonterminating e.g., consider case when program is run with one worker.
1. Deadlock-Free Parallel Program Executions

- A parallel program execution is deadlock-free if no task’s execution remains incomplete due to it being blocked awaiting some condition.

- Example of a program with a deadlocking execution

```java
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
    async await (left) right.put(rightBuilder()); // Task1
    async await (right) left.put(leftBuilder()); // Task2
}
```

- In this case, Task1 and Task2 are in a deadlock cycle.
  - Three constructs that can lead to deadlock in HJ: async await, finish + actors, explicit phaser wait (instead of next)
  - There are many mechanisms that can lead to deadlock cycles in other programming models (e.g., thread join, synchronized, locks in Java)
2. Livelock-Free Parallel Program Executions

- A parallel program execution exhibits livelock if two or more tasks repeat the same interactions without making any progress (special case of nontermination)

- Livelock example:

```java
// Task 1
incrToTwo(AtomicInteger ai) { 
    // increment ai till it reaches 2 
    while (ai.incrementAndGet() < 2);
}
```

```java
// Task 2
decrToNegativeTwo(AtomicInteger ai) { 
    // decrement ai till it reaches -2 
    while (ai.decrementAndGet() > -2);
}
```

- Many well-intended approaches to avoid deadlock result in livelock instead

- Any data-race-free HJ program without isolated/atomic-variables/actors is guaranteed to be livelock-free (may be nonterminating in a single task, however)
3. Starvation-Free Parallel Program Executions

- A parallel program execution exhibits starvation if some task is repeatedly denied the opportunity to make progress
  - Starvation-freedom is sometimes referred to as “lock-out freedom”
  - Starvation is possible in HJ programs, since all tasks in the same program are assumed to be cooperating, rather than competing
    - If starvation occurs in a deadlock-free HJ program, the “equivalent” sequential program must be non-terminating

- Classic source of starvation: “Priority Inversion” problem for OS threads
  - Thread A is at high priority, waiting for result or resource from Thread C at low priority
  - Thread B at intermediate priority is CPU-bound
  - Thread C never runs, hence thread A never runs
  - Fix: when a high priority thread waits for a low priority thread, boost the priority of the low-priority thread
Related Concepts: Progress Condition

- A resource is said to be obstruction-free if it is deadlock-free.
- A resource is said to be lock-free if it is livelock-free and deadlock-free.
- A resource is said to be wait-free if it is starvation-free, livelock-free, and deadlock-free.
  - Question: how to bound the wait duration?
4. Bounded Wait

- A parallel program execution exhibits bounded wait if each task requesting a resource should only have to wait for a bounded number of other tasks to “cut in line” i.e., to gain access to the resource after its request has been registered.

- If bound = 0, then the program execution is fair
A metaphor for Bounded Wait

- **Progress:** If no process is waiting in its critical section and several processes are trying to get into their critical section, then entry to the critical section cannot be postponed indefinitely.

- **Bounded Wait:** A process requesting access to a resource should only have to wait for a bounded number of other processes to access the resource that requested access after it.

A “cut-through” could cause unbounded wait for folks in the loop!