Don’t Start with Dekker’s Algorithm: Top-Down Introduction of Concurrency

Michael L. Scott

Multicore Programming Education Workshop
8 March 2009
Bottom-Up Concurrency

- AKA Concurrency for Wizards
- Usually taught in the OS course
  - Dekker’s algorithm
  - Peterson’s algorithm
  - (maybe) Lamport’s bakery and fast (no contention) locks
  - TAS
  - T&TAS
  - (maybe) MCS
  - semaphores, monitors, (maybe) CCRs
But...

- Where did the threads come from?
- Why do I care? (What are they *for*?)
- Can mere mortals make any of this work?
Concurrency First?

- Sequentiality as a special case
  - See Arvind’s talk after lunch
  - A backlash, perhaps, against concurrency for wizards

- I’m going to suggest an intermediate approach
  - Learn what you need when you need it
  - “Top-down”, but not “concurrency first”
Suggested Principles

- Integrate parallelism & concurrency into the whole curriculum
- Introduce it gradually where it naturally fits
- Provide clear motivation and payoff at each step
- Recognize that
  - everybody needs benefits from multicore
  - many need to deal with events (concurrency)
  - some need to develop concurrent data structures
  - few need to implement synchronization mechanisms or other race-based code
Thinking about Parallelism

- Is it more or less fundamental than sequentiality?
- May be a silly question
  - Dependences among algorithm steps form a partial order
  - I don’t care if you call it
    - a restriction of the empty order
    - or a relaxation of some total order

- Both are ways of thinking about the ordering of algorithmic steps (state transformers)
Concurrency as Control Flow

- My languages text/course talks about
  - sequencing
  - selection
  - iteration
  - procedural abstraction
  - recursion
  - concurrency
  - exception handling and speculation
  - nondeterminacy
Top-Down Concurrency

parallel libraries

deterministic parallelism

explicitly synchronized

event-driven

thread-based

message-based

low-level races
Use
- par-do or spawn/sync w/ compiler-enforced dynamic separation
- speculation in sequential programs
- futures in pure functional languages
- safe futures in impure languages

And maybe
- par-do, spawn/sync, or unsafe futures, w/out enforced separation
- HPF for-all

Consider
- locality
- granularity
- load balance
- design patterns

Straightforward
**Use**

- atomic blocks
- PO-iterators
- loop post-wait
- map-reduce
- condition sync
- locks, monitors, CCRs
- send/receive/rendezvous

**Consider**

- progress
- happens-before
- data race freedom
- 2-phase commit
- consensus, self-stabilization, Byzantine agreement, etc.
Build
- implementation of threads, locks, monitors, transactions, etc.
- nonblocking data structures
- non-DRF algorithms

Consider
- memory models/consistency
- linearizability, serializability
- consensus hierarchy
Where in the Curriculum?

- Computer literacy
- Parallel libraries
- Deterministic parallelism
  - Explicitly synchronized
    - Event-driven
    - Thread-based
    - Message-based
    - Low-level races
- Data structures
- Networks, dist. comp.
- Languages, SW engg., sci. comp.
- OS, arch., par. comp., DBMS
- Graphics, HCI, web computing
Motivation and Rewards

- Need clear payoff, at each step of the way, to motivate further investment/refinement
  - speedup (even if modest, e.g., on 2-core machine)
  - clarity (for event-driven and naturally multithreaded code)

- Will benefit greatly from access to parallel machines
  - Simulators are lousy motivation
  - Niagara boxes are cheap
What Language Do We Use?

- Lamport: This is the wrong question.
  "Imagine an art historian answering ‘how would you describe impressionist painting?’ by saying ‘in French’.”

- MLS: This is the wrong analogy.
  Imagine an art teacher answering “how would you introduce pointillism?” by saying “in oils”.

- Notation matters!
REAL M = (0.0, 0.0);

BEGIN
  M[0] := f(M[0]),  
  M[1] := g(M[1])  
END
static class A implements Runnable {
    double M[];
    A(double m[]) {M = m;}
    public void run () {
        M[0] = f(M[0]);
    }
}

... double M[] = new double[2];
ExecutorService pool = Executors.newFixedThreadPool(2);
pool.execute(new A(M));
pool.execute(new B(M));
pool.shutdown();
try {
    boolean finished = pool.awaitTermination(10, TimeUnit.SECONDS);
} catch (InterruptedException e) { }

static class B implements Runnable {
    double M[];
    B(double m[]) {M = m;}
    public void run () {
        M[1] = g(M[1]);
    }
}
double[] M = new double[2];
Parallel.Do(
    delegate { M[0] = f(M[0]); },
    delegate { M[1] = g(M[1]); }
);

- Where are the other options?
  » production quality (with good IDE)
  » widely used (for practical-minded students)
Summary Recap

- Integrate parallelism & concurrency into the whole curriculum
- Introduce it gradually where it naturally fits
- Provide clear motivation *and payoff* at each step
- Assign projects on real machines
- In real programming languages