Lecture 40: More Concurrency

CSC 131
Fall, 2008

Kim Bruce

Some lecture notes adapted from those of Bob Harper

Java 5: util.concurrent

- Doug Lea utility classes
  - A few general purpose interfaces
  - Implementations tested over several years
- Principal interfaces & implementations
  - Sync -- protocols to acquire and release locks,
    - e.g. Semaphore w/ acquire, release methods
  - BlockingQueue -- classes to insert and delete objects
    - support put, take that block (like bounded buffer)
  - Executor -- executes Runnable tasks
    - You provide control of threads

Java 5 Concurrency Features

```
class BoundedBuffer {   <- array based queue
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
```

Java 5 Concurrency cont.

```
public Object take() throws InterruptedException {
     lock.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length) takeptr = 0;
       --count;
       notFull.signal();
       return x;
     } finally {
       lock.unlock();
     }
   }
 }
```

- Advantage: Separate queues for nonEmpty and nonFull conditions on same lock.

More Message Passing

Ada Tasks

- Synchronous message passing
- Tasks have some features of monitors
- Exports entry names (w/ parameters)
- Entry names have FIFO queues

Accepting an entry

```
select
   [when <cond> =>] <select alternative>
   {or [when <cond> =>] <select alternative>}
   [else <statements>]
end select
```

Caller only blocked in accept, but only one entry can be executed at a time

```
task body Buffer is
    MaxBufferSize: constant INTEGER := 50;
    Store: array(1..MaxBufferSize) of CHARACTER;
    BufferStart: INTEGER := 1;
    BufferEnd: INTEGER := 0;
    BufferSize: INTEGER := 0;
begin
    loop
        select
            when BufferSize < MaxBufferSize =>
                accept insert(ch: in CHARACTER) do
                    Store(BufferEnd) := ch;
                end insert;
                BufferSize := BufferSize + 1;
                BufferEnd := BufferEnd mod MaxBufferSize + 1;
            or when BufferSize > 0 =>
                accept delete(ch: out CHARACTER) do
                    ch := Store(BufferStart);
                end delete;
                BufferSize := BufferSize -1;
                BufferStart := BufferStart mod MaxBufferSize + 1;
            or
                accept more (notEmpty: out BOOLEAN) do
                    notEmpty := BufferSize > 0;
                end more;
            or
                terminate;
        end select;
    end loop
end Buffer;
```

Concurrent ML

- Designed by John Reppy, now U. of Chicago
- Shared memory poor fit for functional langs
  - Message passing
- Threads share dynamically created channels carrying values of arbitrary type
- Threads synchronize by send and receive on channels.

Threads in CML

- New thread created using spawn:
  - val spawn: (unit → unit) → thread_id
- New thread applies function argument to () to begin execution.
  - Terminates when function returns.
  - storage is garbage collected
- Returns unique id for child thread to parent

Channels

- Channels carry values of arbitrary type
  - type ‘a chan
- Created by:
  - val channel: unit → ‘a chan
  - type inferred by use, only carry values of type ‘a
- Unused channels are garbage collected.

Synchronous Send & Receive

- Synchronous ops:
  - val send: ‘a chan * ‘a → unit
  - val recv: ‘a chan → ‘a
- Send blocks its thread until message received
- Recv blocks until matching send occurs
- Synchronize w/ *rendezvous*.

Synchronizing

```
fun child_talk() = let
      val ch = channel()
      val pr = CIO.print
   in
      spawn(fn() => (pr “begin 1\n”; send(ch,0);
                                     pr “end 1\n”));
      spawn(fn() => (pr “begin 2\n”; recv ch;
                                     pr “end 2\n”));
   end;
```

results in

```
begin 1 }  either order
begin 2
end 1   }  either order
end 2
```

Emulate Cell as Thread

- Mutable cell as server accepting requests to set and get value
  - I.e. cell is pair of channels - for request and reply

```
signature CELL = sig
  type ‘a cell
  val new: ‘a -> ‘a cell
  val get: ‘a cell -> ‘a
  val set: ‘a cell * ‘a -> unit
end
```

Mutable Cells as Threads

```
structure Cell :> CELL = struct
   datatype ‘a request = GET | PUT of ‘a

   datatype ‘a cell =
      CELL of {reqCh: ‘a request chan, replyCh: ‘a chan}

   fun new x = ...

   fun get (CELL{reqCh,replyCh}) =
      (send(reqCh, GET); recv(replyCh))

   fun set (CELL{reqCh, replyCh},x) = (send(reqCh, PUT x))
end
```

More

```
fun new x =
   let
      val reqCh = channel()
      val replyCh = channel()
      fun server x =
        (case (recv reqCh) of
           GET => (send(replyCh,x); server x)
         | PUT x’ => server x’)
   in
      (spawn (fn () => server x);
      CELL {reqCh = reqCh, replyCh = replyCh})
   end
```

Observations

- No mutable storage used. State is in recursion
- Request/reply protocol hidden behind CELL abstraction. Can’t accidentally recv from replyCh w/out first sending GET request.
- Synchronous send ensures cell ops are atomic.

Streams as Threads

- Streams can be viewed as suspended computations, producing values only on demand.
- Emulate as threads using send and recv
  - dataflow network

Streams

- Stream of natural numbers

```
fun nats_from start =
  let
    val ch = channel()
    fun loop i = (send(ch,i); loop(i+1))
  in
    spawn(fn () => loop start); ch
  end
```

- recv's on returned channel yield successive nats, starting w/ "start"

Streams

- Filter out multiples of nat in stream

```
fun filter (p,ch) =
  let
    val out = channel()
    fun loop () =
      let
        val i = recv ch
      in
        (if ((i mod p)<>0)
           then send(out,i) else ());loop())
      end
  in
    spawn loop; out
  end
```