CSE-505: Programming Languages

Lecture 20 — Shared-Memory Parallelism and Concurrency

Zach Tatlock

2016

Concurrency and Parallelism

- PL support for concurrency/parallelism a huge topic
  - Increasingly important (not traditionally in PL courses)
  - Lots of active research as well as decades-old work
- We'll just do *explicit threads* plus:
  - Shared memory (*locks* and *transactions*)
  - Futures
  - Synchronous message passing (*Concurrent ML*)
- We'll skip
  - Process calculi (foundational message-passing)
  - Asynchronous methods, join calculus, ...
  - Data-parallel languages (e.g., NESL or ZPL)
  - ...
- Mostly in ML syntax (inference rules where convenient)
  - Even though current OCaml implementation has threads but not parallelism

Concurrency vs. Parallelism

(Terminology not universal, but distinction paramount):

Concurrency *is about correctly and efficiently managing access to shared resources*

- Examples: operating system, shared hashtable, version control
- Key challenge is responsiveness to external events that may arrive asynchronously and/or simultaneously
- Often provide responsiveness via threads
- Often focus on *synchronization*

Parallelism *is about using extra computational resources to do more useful work per unit time*

- Examples: scientific computing, most graphics, a lot of servers
- Key challenge is Amdahl's Law (no sequential bottlenecks)
- Often provide parallelism via threads on different processors and/or to mask I/O latency

Threads

High-level: "Communicating sequential processes"

Low-level: "Multiple stacks plus communication"

From OCaml's `thread.mli`:

```
type t (*thread handle; remember we're in module Thread*)
val create : ('a->'b) -> 'a -> t (* run new thread *)
val self : unit -> t (* what thread is executing this? *)
```

The *code* for a thread is in a closure (with hidden fields) and `Thread.create` actually *spawns* the thread

Most languages make the same distinction, e.g., Java:

- Create a `Thread` object (data in fields) with a `run` method
- Call its `start` method to actually spawn the thread

Why use threads?

One *OR* more of:

1. Performance (multiprocessor *or* mask I/O latency)
2. Isolation (separate errors *or* responsiveness)
3. Natural code structure (1 stack awkward)

It's not just performance

On the other hand, it seems fundamentally harder (for programmers, language implementors, language designers, semanticists) to have multiple threads of execution

One possible formalism (omitting thread-ids)

- Program state is one heap and multiple expressions
- Any e_i might "take the next step" and potentially spawn a thread
- A value in the "thread-pool" is removable
- Nondeterministic with *interleaving granularity* determined by rules

Some example rules for $H; e \to H'; e'; o$ (where $o ::= \cdot \mid e$):

$$\frac{}{H; !l \to H; H(l); \cdot} \qquad \frac{H; e_1 \to H'; e_1'; o}{H; e_1\ e_2 \to H'; e_1'\ e_2; o}$$

$$\frac{}{H; \mathsf{spawn}(v_1, v_2) \to H; 0; (v_1\ v_2)}$$

Formalism continued

The $H; e \to H'; e'; o$ judgment is just a helper-judgment for $H; T \to H'; T'$ where $T ::= \cdot \mid e; T$

$$\frac{H; e \to H'; e'; \cdot}{H; e_1; \ldots; e; \ldots; e_n \to H'; e_1; \ldots; e'; \ldots; e_n}$$

$$\frac{H; e \to H'; e'; e''}{H'; e_1; \ldots; e; \ldots; e_n \to H'; e_1; \ldots; e'; \ldots; e_n; e''}$$

$$\frac{}{H; e_1; \ldots; e_{i-1}; v; e_{i+1}; \ldots; e_n \to H; e_1; \ldots; e_{i-1}; e_{i+1}; \ldots; e_n}$$

Program termination: $H; \cdot$

Equivalence just changed

Expressions equivalent in a single-threaded world are not necessarily equivalent in a multithreaded context!

Example in OCaml:

```
let x, y = ref 0, ref 0 in
create (fun () -> if (!y)=1 then x:=(!x)+1) ();
create (fun () -> if (!x)=1 then y:=(!y)+1) () (* 1 *)
```

Can we replace line (1) with:

```
create (fun () -> y:=(!y)+1; if (!x)<>1 then y:=(!y)-1) ()
```

For more compiler gotchas, see "Threads cannot be implemented as a library" by Hans-J. Boehm in PLDI2005

- Example: C bit-fields or other adjacent fields

Communication

If threads do nothing other threads need to "see," we are done

- Best to do as little communication as possible
- E.g., do not mutate shared data unnecessarily, or hide mutation behind easier-to-use interfaces

One way to communicate: Shared memory

- One thread writes to a ref, another reads it
- Sounds nasty with pre-emptive scheduling
- Hence synchronization mechanisms
  - Taught in O/S for historical reasons!
  - Fundamentally about restricting interleavings

Join

"Fork-join" parallelism a simple approach good for "farm out subcomputations then merge results"

```
(* suspend caller until/unless arg terminates *)
val join : t -> unit
```

Common pattern:

```
val fork_join : ('a -> 'b array) -> (* divider *)
                ('b -> 'c) ->       (* conqueror *)
                ('c array -> 'd) -> (* merger *)
                'a ->               (* data *)
                'd
```

Apply the second argument to each element of the `'b array` in parallel, then use third argument *after* they are done

See `lec20code.ml` for implementation and related patterns (untested)

Futures

A different model for explicit parallelism without explicit shared memory or message sends

- Easy to implement on top of either, but most models are easily inter-implementable
- See ML file for implementation over shared memory

```
type 'a promise;
val future : (unit -> 'a) -> 'a promise (*do in parallel*)
val force : 'a promise -> 'a (*may block*)
```

Essentially fork/join with a value returned?

- Returning a value more functional
- Less structured than “cobegin s1; s2; … sn” form of fork/join

Locks (a.k.a. mutexes)

```
(* mutex.mli *)
type t (* a mutex *)
val create : unit -> t
val lock   : t -> unit (* may block *)
val unlock : t -> unit
```

OCaml locks do not have two common features:

- Reentrancy (changes semantics of `lock` and `unlock`)
- Banning nonholder release (changes semantics of `unlock`)

Also want condition variables (`condition.mli`), not discussed here

Using locks

Among infinite correct idioms using locks (and more incorrect ones), the most common:

- Determine what data must be "kept in sync"
- Always acquire a lock before accessing that data and release it afterwards
- Have a *partial order* on all locks and if a thread holds m_1 it can acquire m_2 only if $m_1 < m_2$

See canonical "bank account" example in `lec20code.ml`

Coarser locking (more data with same lock) trades off parallelism with synchronization

- Under-synchronizing the hallmark of concurrency incorrectness
- Over-synchronizing the hallmark of concurrency inefficiency

The Evolution Problem

Write a new function that needs to update $o1$ and $o2$ together.

- What locks should you acquire? In what order?
- There may be no answer that avoids *races* and *deadlocks* without breaking old code. (Need a stricter partial order.)
  - Race conditions: See definitions later in lecture
  - Deadlock: Cycle of threads blocked forever

See `xfer` code in `lec16code.ml`

Real Java example:

```
synchronized append(StringBuffer sb) {
 int len = sb.length(); //synchronized
 if(this.count+len > this.value.length) this.expand(...);
 sb.getChars(0,len,this.value,this.count); //synchronized
 ...
}
```

Undocumented in 1.4; in 1.5 caller synchronizes on `sb` if necessary

Atomic Blocks (Software Transactions)

Java-like: `atomic { s }`

OCaml-like: `atomic : (unit -> 'a) -> 'a`

Execute the body/thunk *as though* no interleaving from other threads

- Allow parallelism unless there are actual run-time memory conflicts (detect and abort/retry)
- Convenience of coarse-grained locking with parallelism of fine-grained locking (or better)
- But language implementation has to do more to detect conflicts (much like garbage collection is convenient but has costs)

Most research on implementation (preserve parallelism unless there are conflicts), but this is not an implementation course

Transactions make things easier

Problems like `append` and `xfer` become trivial

So does mixing coarse-grained and fine-grained operations (e.g., hashtable lookup and hashtable resize)

Transactions *are* great, but not a panacea:

- Application-level races can remain
- Application-level deadlock can remain
- Implementations generally try-and-abort, which is hard for "launch missiles" (e.g., I/O)
- Many software implementations provide a weaker and under-specified semantics if there are data races with non-transactions
- *Memory-consistency model* questions remain and may be worse than with locks...