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 Caml 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

### 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
Formalism continued

The $H; e \rightarrow H'; e'$ judgment is just a helper-judgment for $H; T \rightarrow H'; T'$ where $T ::= \cdot | e; T$

$H; e \rightarrow H'; e'$

$H; e_1; \ldots; e; e_n \rightarrow H'; e_1; \ldots; e; e_n$

Equivalence just changed

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

Example in Caml:

```
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 lec19.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
```

Caml 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), but skipping
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 lec19.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

Getting it wrong
Races can result from too little synchronization
- Data races: simultaneous read-write or write-write of same data
  - Lots of PL work in last 15 years on types and tools to prevent/detect
  - Provided language has some guarantees, may not be a bug
    - Canonical example: parallel search and “done” bits
    - But few language have such guarantees (!)
- Higher-level races: much tougher to prevent in the language
  - Amount of correct nondeterminism inherently app-specific

Deadlock can result from too much synchronization
- Cycle of threads waiting for someone else to do something
- Easy to detect dynamically with locks, but then what?

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.)

See xfer code in lec19.ml, which is yet another binary-method problem for OOP.

Real example from Java:
```java
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 }
Camllike: 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...

Memory models
A memory-consistency model (or just memory model) for a concurrent shared-memory language specifies “which write a read can see”.

The gold standard is sequential consistency (Lamport): “the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”

Under sequential consistency, this assert cannot fail, despite data races:
```ocaml
let x, y = ref 0, ref 0
let _ = create (fun () -> x := 1; y := 1) ()
let _ = create (fun () -> let r = !y in let s = !x in assert(s>r) ()
```

Relaxed memory models

Modern imperative and OO languages do not promise sequential consistency (if they say anything at all)

- The hardware makes it prohibitively expensive
- Renders unsound almost every compiler optimization

Example: common-subexpression elimination

Initially a==0 and b==0
Thread 1          Thread 2
x=a+b;          b=1;
y=a;            a=1;
z=a+b;
assert(z>=y);

In real languages

- Java: If every sequentially consistent execution of program P is data-race free, then every execution of program P is equivalent to some sequentially consistent execution
  - Not the definition, a theorem about the definition
  - Actual definition very complicated, balancing needs of code writers, compiler optimizers, and hardware
    - Complicated by constructors and final fields
    - Not defined in terms of “list of acceptable optimizations”

- C++: Roughly, any data race is as undefined as an array-bounds error. No such thing as a benign data race and no guarantees if you have one. (In practice, programmers will still assume things, like they do with casts.)
  - But same theorem as Java: “DRF ⇒ SC”

- Most languages: Eerily silent
  - IMO, the greatest current failure of programming languages

Mostly functional wins again

If most of your data is immutable and most code is known to access only immutable data, then most code can be optimized without any concern for the memory model

So can afford to be very conservative for the rest

Example: A Caml program that uses mutable memory only for shared-memory communication

Non-example: Java, which uses mutable memory for almost everything
  - Compilers try to figure out what is thread-local (again avoids memory-model issues), but it’s not easy

Ordering and atomic

Initially x==0 and y==0
Thread 1          Thread 2
x=1;            r=y;
y=1;            s=x;
Can s be less than r?

Yes.

Ordering and atomic

Initially x==0 and y==0
Thread 1          Thread 2
x=1;            r=y;
sync(lk){}    sync(lk){}
y=1;            s=x;
Can s be less than r?

In Java, no.
  - Notion of “happens-before” ordering between release and acquire of the same lock
Ordering and atomic

Initially x==0 and y==0

Thread 1                  Thread 2
x=1;                      r=y;
atomic{}                  atomic{}
y=1;                      s=x;

Can s be less than r?

Nobody really knows, but often yes (!) in prototype implementations