Welcome!

1st of 32 lectures (4/day * 10 days = 32 ☺)
- As an introduction, different than most

• A few minutes on the school, you, etc.
• A few minutes on why language-based concurrency
• Some lambda-calculus and naïve concurrency
• Rough overview of what the school will cover

I get 2 lectures next week on software transactions
- Some of my research

A simple plan

• 11 speakers from 9 institutions
• “36” of you (28 PhD students, 5 faculty, 3 industry)
• Lectures at a PhD-course level
  - More tutorial/class than seminar or conference
  - Less homework and cohesion than a course
  - Not everything will fit everyone perfectly
    • Early stuff more theoretical
• Advice
  - Make the most of your time surrounded by great students and speakers
  - Be inquisitive and diligent
  - Have fun

Why concurrency

PL summer school not new; concurrency focus is
1. Concurrency/distributed programming now mainstream
   • Multicore
   • Internet
   • Not just scientific computing
2. And it’s really hard (much harder than sequential)
3. There is a lot of research (could be here 10 months)
4. A key role for PL to play...

Thanks!

• Jim: none of us would be here without him
• Jeff: the co-organizer
• Steering committee
  - Zena Ariola, David Walker, Steve Zdancewic
• Sponsors
  - Intel
  - National Science Foundation
  - Google
  - ACM SIGPLAN
  - Microsoft

Why PL

“what does it mean for computations to happen at the same time and/or in multiple locations”
“how can we best describe and reason about such computations”

Biased opinion: Those are PL questions and PL has the best intellectual tools to answer them
• “Learn concurrency in O/S class” a historical accident that will change soon
Why do people do it

If concurrent/distributed programming is so difficult, why do it?
• Performance
  (exploit more resources; reduce data movement)
• Natural code structure
  (independent communicating tasks)
• Failure isolation (task termination)
• Heterogeneous trust (no central authority)
  *It's not just "parallel speedup"*

Outline

1. Lambda-calculus / operational semantics tutorial
2. Naively add threads and mutable shared-memory
3. Overview of the much cooler stuff we'll learn
   *“Starting with sequential” is only one approach*
Remember this is just a tutorial/overview lecture
• No research results in the next hour

Lambda-calculus in n minutes

• To decide “what concurrency means” we must start somewhere
• One popular sequential place: a lambda-calculus
• Can define:
  – Syntax (abstract)
  – Semantics (operational, small-step, call-by-value)
  – A type system (filter out “bad” programs)

Syntax

Syntax of an untyped lambda-calculus
Expressions: \( e ::= x \mid \lambda x. e \mid e \ e \mid c \mid e + e \)
“Constants: \( c ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \)”
“Variables: \( x ::= x \mid y \mid x_1 \mid y_1 \mid \ldots \)”
Values: \( v ::= \lambda x. e \mid c \)
Defines a set of trees (ASTs)
Conventions for writing these trees as strings:
• \( \lambda x. e_1 \ e_2 \) is \( \lambda x. \ (e_1 \ e_2) \), not \( (\lambda x. e_1) \ e_2 \)
• \( e_1 \ e_2 \ e_3 \) is \( (e_1 \ e_2) \ e_3 \), not \( e_1 \ (e_2 \ e_3) \)
• Use parentheses to disambiguate or clarify

Semantics

• One computation step rewrites the program to something “closer to the answer”
  \( e \rightarrow e' \)
• Inference rules describe what steps are allowed

Notes

• These are rule schemas
  – Instantiate by replacing metavariables consistently
• A derivation tree justifies a step
  – A proof: “read from leaves to root”
  – An interpreter: “read from root to leaves”
• Proper definition of substitution requires care
• Program evaluation is then a sequence of steps
  \( e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots \)
• Evaluation can “stop” with a value (e.g., 17) or a “stuck state” (e.g., \( 17 \ \lambda x. \ x \))
More notes

- I chose left-to-right call-by-value
- Easy to change by changing/adding rules
- I chose to keep evaluation-sequence deterministic
- Also easy to change; inherent to concurrency
- I chose small-step operational
- Could spend a year on other semantics
- This language is Turing-complete (even without constants and addition)
- Therefore, infinite state-sequences exist

Outline

1. Lambda-calculus / operational semantics tutorial
2. Naively add threads and mutable shared-memory
3. Overview of the much cooler stuff we’ll learn

“Starting with sequential” is only one approach

Remember this is just a tutorial/overview lecture
- No research results in the next hour

Types

A 2nd judgment \( \Gamma \vdash e : t \) gives types to expressions
- No derivation tree means “does not type-check”
- Use a context to give types to variables in scope

“Simply typed lambda calculus” a starting point

Types: \( t ::= \text{int} \mid t \to t \)

Contexts: \( \Gamma ::= \cdot \mid \Gamma, x : t \)

\[
\begin{array}{c}
\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma, x : t \vdash e_1 + e_2 : t \\
\Gamma, (\lambda x. e) : t_1 \to t_2 \vdash e_1 : t_1 \\
\Gamma \vdash e_1 : t_2 \\
\end{array}
\]

Adding concurrency

- Change our syntax/semantics so:
  - A program-state is \( n \) threads (top-level expressions)
  - Any one might “run next”
  - Expressions can fork (a.k.a. spawn) new threads

Expressions: \( e ::= \_ \mid \text{fork } e \)
States: \( P ::= \_ \mid e : P \)
Exp options: \( o ::= \text{None} \mid \text{Some } e \)

Change \( e \to e' \) to \( e \to e', o \)
Add \( P \to P' \)

Semantics

\[
\begin{array}{l}
\begin{array}{ll}
e_1 \to e_1', \_ & e_2 \to e_2', \_ \\
e_1 e_2 \to e_1' e_2, o & v e_2 \to v e_2', o \\
\end{array} \\
e_1 \to e_1', o & (\lambda x. e) v \to e(v), \_ \\
\begin{array}{ll}
e_2 \to e_2', o \\
v e_2 \to v e_2', o \\
\end{array} \quad \begin{array}{ll}
\_ & c_1 + c_2 = c_3 \\
\_ & v_1 e_2 = v_1 e_2', o \\
\_ & c_1 + c_2 = c_3, \_ \\
\end{array} \\
\begin{array}{ll}
e_1 + e_2 \to e_1' + e_2, o & v_1 e_2 \to v_1 e_2', o \\
\_ & c_1 + c_2 = c_3, \_ \\
\_ & v_1 e_2 \to v_1 e_2', o \\
\end{array} \\
\begin{array}{ll}
\text{fork } e \to e_2, \text{Some } e \\
\end{array} \quad \begin{array}{ll}
e_1 \to e_1', \_ \\
\text{fork } e \to e_2, \text{Some } e \\
\end{array} \\
\begin{array}{ll}
\_ & \text{None} \\
\_ & \text{Some } e \\
\end{array} \\
\begin{array}{ll}
\_ & e_1 \to e_1', \_ \\
\_ & e_1 \to e_1', \text{Some } e \\
\_ & e_1 \to e_1', \_ \\
\end{array} \\
\end{array}
\]

Notes

In this simple model:
- At each step, exactly one thread runs
- “Time-slice” duration is “one small-step”
- Thread-scheduling is non-deterministic
- So the operational semantics is too?
- Threads run “on the same machine”?
- A “good final state” is some \( v_1 \ldots ; v_n ; \_ \)
- Alternately, could remove done threads: 

\[
e_1 \ldots ; e_1 ; v_1 ; e_2 ; \_ ; \ldots e_n ; \_ 
\]
Not enough

- These threads are really uninteresting; they can’t communicate
- One thread’s steps can’t affect another
- All final states have the same values
- One way: mutable shared memory
- Many other communication mechanisms to come!
- Need:
  - Expressions to create, access, modify mutable locations
  - A map from mutable locations to values in our program state

Changes to old stuff

Expressions: e ::= ... | ref e | e :: e2 | `e | 1
Values: v ::= ... | 1
Heaps: H ::= ... | H, l := v
Thread pools: P ::= ... | e : P
States: H, P
Change e -> e’, o to H, e -> H’, e’, o
Change P -> P’ to H, P -> H’, P’
Change rules to modify heap (or not). 2 examples:

<table>
<thead>
<tr>
<th>H, e1, e1 -&gt; H’, e1’, e2, o</th>
<th>“c1 + c2 = c3”</th>
</tr>
</thead>
<tbody>
<tr>
<td>H, e2, e2 -&gt; H’, e2’, o</td>
<td>H, c1 + c2 -&gt; H, c3, None</td>
</tr>
</tbody>
</table>

New rules

1 not in H

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>H, e1 := e -&gt; H, e1 := e, 42, None</td>
<td></td>
</tr>
<tr>
<td>H, e := e -&gt; H’, e’, o</td>
<td></td>
</tr>
<tr>
<td>H, e =&gt; H’, e’, o</td>
<td></td>
</tr>
<tr>
<td>H, e, e := e1, e1’, e1’ := e2, o</td>
<td></td>
</tr>
<tr>
<td>H, e, e2 := e2, e2’, o</td>
<td></td>
</tr>
</tbody>
</table>

Now we can do stuff

We could now write “interesting examples” like
- Fork 10 threads, each to do a different computation
- Have each add its answer to an accumulator 1
- When all threads finish, 1 is the answer

Problems:
1. If this is not the whole program, how do you know when all 10 threads are done?
   - Solution: have them increment another counter
2. If each does 1 ::= !1 + e, there are races...

Races

1 ::= !1 + 35
An interleaving that produces the wrong answer:
Thread 1 reads 1
Thread 2 reads 1
Thread 1 writes 1
Thread 2 writes 1 – “forgets” thread 1’s addition
Communicating threads must synchronize
Languages provide synchronization mechanisms, e.g., locks...

Locks

Two new expression forms:
- acquire e
  - if e is a location holding 0, make it hold 1
  - (else block: no rule applies; thread temporarily stuck)
  - (test-and-set is atomic)
- release e
  - same as e := 0; added for symmetry
Adding formal inference rules: “exercise”
Using this for our example: “exercise”
Adding condition variables: “more involved exercise”
Locks are hard

Locks can avoid races when properly used
• But it’s up to the programmer
• And “application-level races” may involve multiple locations
  – Example: “11 > 0 only if 12 = 17”

Locks can lead to deadlock
Trivial example:
  - acquire i1 acquire i2
  - acquire i2 acquire i1
  - release i2 release i1
  - release i1 release i2

Summary

We added
1. Concurrency via fork and non-deterministic scheduling
2. Communication via mutable shared memory
3. Synchronization via locking
There are better models; this was almost a “straw man”

Even simple concurrent programs are hard to get right
  – Races and deadlocks common
And this model is much simpler than reality
  – Distributed computing; relaxed memory models

Outline

1. Lambda-calculus / operational semantics tutorial
2. Naively add threads and mutable shared-memory
3. Overview of the much cooler stuff we’ll learn
  “Starting with sequential” is only one approach
Remember this is just a tutorial/overview lecture
  • No research results in the next hour

Some of what you will see

1. Richer foundations (theoretical models)
2. Dealing with more complicated realities
3. Other communication/synchronization primitives
4. Techniques for improving lock-based programming
  [This is not in the order we will see it]

Foundations

• Process-calculi [Sewell]
  – Inherently parallel (rather than an add-on)
  – Communication over channels

• Modal logic [Harper]
  – Non-uniform resources
  – Types for distributed computation

• Provably efficient job scheduling [Leiserson/Kuszmaul]
  – Optimal algorithms for load-balancing

Realities

• Distributed programming [Sewell] [Harper]
  – Long latency, lost messages, version mismatch, ...

• Relaxed memory models [Dwarkadas]
  – Hardware does not give globally consistent memory

• Dynamic software updating [Hicks]
  – Cannot assume fixed code during execution

• Termination [Fiatt]
  – Threads may be killed at inopportune moments
Ways to synchronize, communicate

- Fork-join [Leiserson/Kuszmaul]
  - Block until another computation completes
- Futures [Hicks]
  - Asynchronous calls (less structured fork/join)
- Message-passing a la Concurrent ML [Flatt]
  - First-class synchronization events to build up communication protocols
- Software transactions, a.k.a. atomicity…

Atomicity

An easier-to-use and harder-to-implement synchronization primitive:

\[
\text{atomic} \{ \ s \ \}
\]

Must execute $s$ as though no interleaving, but still ensure fairness.

- Language design & software-implementation issues [Grossman]
- Low-level software & hardware support [Dwarkadas]
- As a checked/inferred annotation for lock-based code [Flanagan]

Analyzing lock-based code

- Type systems for data-race and atomicity detection [Flanagan]
  - Static & dynamic enforcement of locking protocols
- Analysis for multithreaded C code; “what locks what” [Foster]
  - Application to systems code; incorporating alias analysis
- Model-checking concurrent software [Qadeer]
  - Systematic state-space exploration

Some of what you will see

1. Richer foundations (theoretical models)
2. Dealing with more complicated realities
3. Other communication/synchronization primitives
4. Techniques for improving lock-based programming

[This is not in the order we will see it]

Thanks in advance for a great summer school!