Announcements

- HW9: Java Concurrency
  - Test cases up in Homework Server
  - Due May 9th. Extended to Wed May 11th 6pm

- Rainbow grades coming up tomorrow
  - HW 1-7
  - Exam 1-2
  - Quiz 1-9
  - Questions/concerns, contact us ASAP

Last Class

- Java concurrency assignment
  - Consistency with two-phase locking
  - Avoiding deadlock

- Concurrency in Java
  - Threads and Locks
  - What can go wrong?
    - Data races, atomicity errors, deadlocks
  - More on synchronization

Today’s Lecture Outline

- Concurrency… Conclusion
  - Memory consistency models
  - What are some developments in the area

- Scripting/Dynamic Programming Languages
- Taking Stock: PL Design Choices
- Python

Memory Consistency Models

- When more than one location is written at about the same time, we must worry about the order the writes become visible to different processors

- Sequential consistency is the intuitive model
  - We want all writes to become visible at the same time to all processors
  - We want a given processor’s writes to be visible in the order they were performed
  - Sequential consistency is so hard, that processors simply do not provide it!

Memory Consistency Models (Sc. p. 611)

Initially: \texttt{inspected = false; X = 0}
Processor A Processor B
\texttt{inspected := true X := 1}
\texttt{xa := X ib := inspected}

Processor A and Processor B execute their respective code at about the same time. What values do you expect for \texttt{xa} and \texttt{ib}?

Memory Consistency Models (Sc. p. 611)

Initially: \texttt{inspected = false; X = 0}
Processor A Processor B
\texttt{inspected := true X := 1}
\texttt{xa := X ib := inspected}

In fact, processors buffer writes and they don’t become visible to other processors right away! It is possible to get \texttt{xa = 0} and \texttt{ib = false}!
**volatile keyword in Java**

- `volatile x` forces reads and writes on `x` to be ordered across threads.
  - Intuitively, a read of `x` (by one processor) will see the most recently written value (by different one). Buffering is not allowed on `volatile` variables.
- How can we “fix” our example?

Initially: `inspected = false; X = 0`

Thread A
```
inspected := true
```
```
xa := X
```

Thread B
```
X := 1
```
```
ib := inspected
```

---

**Java Memory Model**

- A thread can buffer its writes until it writes a `volatile` variable or exits a synchronized block.
- A thread can use cached (buffered) values until it reads a `volatile` variable or enters a synchronized block.
- Rules ensure sequential consistency for data-race-free programs.

---

**Concurrency in Java**

- Most of the time we synchronize with mutex locks and `volatile` variables.
  - Increasingly programmers chose alternatives such as CAS (compare-and-swap)-based non-blocking structures and algorithms, etc.
  - Programming with these alternatives is even harder!

---

**Concurrency. What can be done?**

- Programming languages
  - Transactional memory
  - Actor languages
- Software tools
  - Dynamic tools – monitor program at runtime
  - Static tools – examine program code and try to reason about what can happen at runtime

---

**Transactional Memory**

- Premise: locks are too low-level.
- Basic idea, borrows from Database Transactions: programmer specifies atomic blocks. The system ensures that the atomic block is executed at once.
  - Programmer can think sequentially: `atomic { i++; }
Thread A:
```
atomic {
    r1 = i;
    r1 += 1;
    i = r1;
}
```
Thread B:
```
atomic {
    r2 = i;
    r2 += 1;
    i = r2;
}
```

---

**Transactional Memory**

- Uses the optimistic concurrency model.
  - Assume no conflicts and execute... Monitor for conflicts at runtime, if conflict, rollback.
- What we need is a smart runtime/threads that
  - Detect conflicts
  - Support commits, rollbacks and retries
    - If there are no conflicts, commit “atomic block”
    - Otherwise, roll back and retry later
- Why is this fundamentally more difficult to accomplish for programs than for database transactions?
Problems?

- What do we do with IO? Not easy to roll back…
- Clearly, a data race that involves two atomic sections is a conflict. But what if there is a data race between an atomic section and code outside of an atomic section?
  - Weak atomicity vs. strong atomicity
- What if the code in the atomic section throws an exception?

More Problems?

- What is the granularity of atomic sections
  - Too large sections sacrifice concurrency
  - Too small sections leave race conditions and atomicity violations
  - How do we discourage the programmer from writing too large atomic sections?
- Performance – software transactional memory is still much much slower than explicit synchronization

Actor Languages

- Actor languages implement message-passing concurrency … on a shared-memory machine
  - Support for actors --- typically mapped to threads
    - Actors encapsulate state
    - Communicate with messages
  - No shared memory! Thus, no data races!
- Erlang, Scala, SALSA
- Pros and Cons of Actor Languages?

Software Tools

- Dynamic software tools --- monitor program execution and detect race conditions, atomicity violations, deadlocks, etc. at runtime
- Dynamic data race detectors are popular
  - Happens-before-based race detectors
  - Lockset-based race detectors

Happens-before Dynamic Detectors

- Most popular dynamic race detectors are based on the happens-before ordering relation
- Happens-before is a partial order on all thread operations
  - Within a thread, operations are ordered in the order that they occur
  - Between threads, operations are ordered, if they are synchronized with one another

Happens-before: Example

Thread A:

```
lock(o)
rl = v
rl += 1
v = rl
unlock(o)
```

Thread B:

```
lock(o)
r2 = v
r2 += 1
v = r2
unlock(o)
```
Happens-before: Another Example

Thread A:

\[ \begin{align*}
    & y = y + 1 \\
    & \text{lock (o)} \\
    & v = v + 1 \\
    & \text{unlock (o)}
\end{align*} \]

Thread B:

\[ \begin{align*}
    & \text{lock (o)} \\
    & v = v + 1 \\
    & \text{unlock (o)} \\
    & y = y + 1
\end{align*} \]

Happens-before: Dynamic Detection

- If two threads access a shared variable, where at least one access is a write, and the accesses are not ordered by happens-before, the detector reports a data race.
- Happens-before-based tools monitor each data reference and check for accesses that are not ordered by happens-before.

Problems?

- Very difficult to implement
- Engineering issues
- Efficiency: happens-before detectors have huge overhead 10x to 30x original execution time
- Race reports depend on scheduling of interleaving events!

Lockset-based Dynamic Detectors

- Late 90’s. Dynamic race detectors based on the lockset algorithm
- Savage et al., ACM TOPLAS, Vol. 15, No. 4, Nov. 97, Eraser: A Dynamic Data Race Detector for Multithread Programs
- Improves on happens-before

Lockset-based Dynamic Detectors

- Race detector maintains lockset \( C(v) \) for each shared variable \( v \)
  - Lockset is initialized to all locks and refined as program executes. If lockset becomes empty, issue a warning.

Let \( \text{locks\_held}(t) \) be the locks held by thread \( t \)

Initialize \( C(v) \) to the set of all locks
On each access to \( v \)
\[ C(v) = C(v) \cap \text{locks\_held}(t) \]
if \( C(v) \) is empty issue a warning.
Problems?
- Efficiency: huge overhead
- False positives are possible!
- There are many variations of lockset-based race detectors

Race Detectors
- Some publicly available race detectors
  - ThreadSanitizer
    - A Valgrind (?) happens-before dynamic race detector for C/C++
  - Chord
    - A static race detector for Java

Today’s Lecture Outline
- Concurrency… Conclusion
- Scripting/Dynamic Programming Languages
  - Taking Stock: PL Design Choices
  - Python

Scripting Languages
- E.g., Tcl, awk
- Originate in the 1970’s from UNIX (shell scripts)
- Purpose
  - To process text files with ease
  - To launch components and combine (or “glue”) these components into an application
- Characteristics
  - Ease of use, flexibility, rapid prototyping: hence, scripting languages are dynamically typed
  - Extensive support for text processing

References
- Slides 9-18 are based on
  - “The Rise of Dynamic Languages” by Jan Vitek
  - “The Essence of JavaScript”, ECOOP’10 paper by Guha et al.

2 Decades of Dynamic Languages
- Visual Basic dyn 1991
- Python dyn 1991
- Lua dyn 1993
- R dyn 1993
- Java stat+dyn 1995
- JavaScript dyn 1995
- Ruby dyn 1995
- PHP dyn 1995
- C# stat+dyn 2001
- Scala stat 2002
- F# stat 2003
- Clojure dyn 2007
- Last decade:
  - Go stat
  - Rust stat
  - Swift stat
Characteristics

- **Dynamic typing**, also known as Duck typing
  - Type checking amounts to running the “Duck test” at runtime:
    
    "If it walks like a duck and swims like a duck and quacks like a duck, then it is a duck."  --- paraphrased from J. W. Riley

    ```javascript
    fun F(x) {
        x.quack();
    }
    ```

    - Don’t use `eval`, unless absolutely necessary

Other Characteristics

- Reference model, Garbage collected
- Reflective! (i.e., `eval` is a prominent feature)
  - Use of `eval` ranges from sensible to stupid (ref. “The eval that men do” by Richards et al.)
    ```javascript
    var flashVersion = parse();
    flash2Installed = flashVersion == 2;
    ... if same for 3 to 11
    for (var i = 2; i <= maxVersion; i++)
        if (eval("flash"+i+"Installed")==true)
            actualVersion = i;
    Don’t use `eval`, unless absolutely necessary
    ```

Objects and Fields in JavaScript

- **Objects** have **fields** (field are also known as properties)
- Field lookup
  ```javascript
  x["f"], x.f
  { "f" : 7 }["g"]
  ```
  
  JavaScript is **error-oblivious**, in the above example, it returns `undefined`, and continues!

- Field update
  ```javascript
  x["f"] = 2, x.f = 2
  { "f" : 0 }["g"] = 10 yields
  { "f" : 0, "g" : 10 }
  ```

A bit on JavaScript

- 91% of the top 10K web pages
- Single-threaded
- Reference model, Garbage collected
- Reflective! (i.e., `eval` is a prominent feature)
- High-level data structures, libraries
- Lightweight syntax
- Error-oblivious
Objects and Fields in JavaScript

- Field delete
  - `delete x.f`
  - `delete { "f" : 7, "g" : 0 }["g"]`

Arrays Are Objects

```javascript
function sum(arr){
    var r = 0;
    for (var i=0; i<arr["length"]; i=i+1){
        r = r + arr[i]
    }
    return r
}
sum([1,2,3]) yields what?
var a = [1,2,3,4];
delete a["3"];
sum(a) yields what?
```

Functions Are Objects

```javascript
f = function(x) { return x+1 }
f.y = 90
f(f.y) yields what?
```

Other unexpected behaviors...

```javascript
with statement
    eval
See https://www.destroyallsoftware.com/talks/wat
```

Taking Stock: Design Choices

<table>
<thead>
<tr>
<th></th>
<th>Datatypes</th>
<th>Control-Flow</th>
<th>Basic Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme</td>
<td>Numbers,</td>
<td>Conditional</td>
<td>Function</td>
</tr>
<tr>
<td></td>
<td>Symbols,</td>
<td>flow, Recursion</td>
<td>application</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>Primitive</td>
<td>Conditional</td>
<td>Assignment</td>
</tr>
<tr>
<td></td>
<td>types, Array</td>
<td>flow, Iteration</td>
<td>statement</td>
</tr>
<tr>
<td></td>
<td>types, Classes (library, and user-defined)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>Primitive</td>
<td>Conditional</td>
<td>Assignment</td>
</tr>
<tr>
<td></td>
<td>types, struct types, pointer types, array types, classes</td>
<td>flow, Iteration</td>
<td>statement</td>
</tr>
</tbody>
</table>

Python

- Designed by Guido van Rossum at CWI Amsterdam in 1991
- Multi-paradigm programming language
  - All characteristics of dynamic languages
  - It has “functional” features
    - E.g. higher-order functions, map and reduce on lists, list comprehensions
  - It has “object-oriented” features
    - E.g., iterators, array slicing operations, reflection, exceptions, (multiple) inheritance, dynamic loading
### Python: Syntax and Scoping

Variable belongs to block where it is written, unless explicitly imported.

What is the output of this program?

```python
m=1
j=3
def outer():
    def middle(k):
        def inner():
            global m;
            # from main program, not outer
            m=4;
            print m,j,k
        inner()
        return m,j,k  # 3 element tuple
    m=2;
    # new local m
    return middle(j)  # old (global) j
What’s printed if we removed the red “global m” statement? then remove the blue assignment?
```

### Scoping: Common Indentation Error

```bash
~/python> python scope.py
File "scope.py", line 10
  m=2;             #new local m
^IndentationError: unindent does not match any outer indentation level
```

### Scoping: Static Scoping Rules

- Blocks (scopes) are defined by indentation
- There are no variable declarations
  - A variable belongs to block where it is written
- “Closest nested scope” rule applies
  - Lookup proceeds from inner to outer blocks
  - ‘global’ overrides rule for access to outermost scope

### Datatypes

- Numbers (+, *, **, pow, etc) - immutable
- Collections
  - Sequences
  - Strings
  - Lists
  - Tuples
- Mappings
  - Dictionaries
- Files

### Datatypes

- So, what model for variables does Python use?
- Reference model for variables

### Equality

- `==` equal in value (structural equality, value equality)
- `is` same object (reference equality)
- `None` acts like null in C, a placeholder for an actual value

### Control Flow

- Short-circuit evaluation of boolean expressions
- Constructs for conditional control flow and iteration: if, while, for
  ```python
  for x in ["spam", "eggs", "ham"]:  
    print x # iterates over list elements
  for y in s: print y # iterates over chars
  ```
- Use of iterators defines the “Python style”
Functions

Two forms of function definition

```python
def incr(x):
    return x+1  # function incr

incrs = [lambda x: x+1, lambda x: x+2]
```

Polymorphism

# what datatypes can this function be used on?
```python
def intersect(seq1, seq2):
    res = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
```

- Call by value
  - But each value is a reference!
  - So we say that parameter passing is call by sharing
  - Be careful: if we pass a reference to a mutable object, callee may change argument object

Taking Stock: Python

- Datatypes?
- Control flow?
- Basic operation?
- Variable model?
- Parameter passing mechanism?
- Scoping?
  - Are functions first-class values?
- Typing?