Announcements

- Quiz 9
- HW12: Transaction Server  
  - Due Tuesday December 11th
- Rainbow grades: HW1-9, Quiz1-7, Exam1-2  
- Still grading: HW10 and HW11
- Grad students enrolled in CSCI-6430:  
  - Paper critique assignment sent to your rpi email
  - Send me your write-up by email

Last Class

- Transaction Server (HW12)  
  - Overview
  - Consistency with two-phase locking
- What can go wrong?
- Data race detection
- Alternative programming models: Actor Model

Today’s Lecture Outline

- Concurrency… Conclusion
  - Producer Consumer in Java and Scala
  - Memory consistency models
- Scripting/Dynamic Programming Languages
- Taking Stock: PL Design Choices
- Python

Producer Consumer

- Classical concurrency problem
  - Producers produce items in shared buffer
  - Consumers consume items from buffer
  - Each producer, consumer runs in its own thread

Solution in Java: BoundedBuffer

```java
class BoundedBuffer {
    private int numSlots;  // size of buffer (queue)
    private int[] buffer;  // buffer (queue)
    private int takeOut = 0, putIn = 0;  // front (takeOut) and back (putIn) of circular queue
    private int count=0;  // number of items currently in queue

    public synchronized void put(int value)  
        throws InterruptedException {
            while (count == numSlots)  
                wait();  // blocks current thread
            buffer[putIn] = value;
            putIn = (putIn + 1) % numSlots;
            count++;
            notifyAll();  // notifies all threads
        }
}
```

BoundedBuffer.put (i.e., produce)
public synchronized int get() throws InterruptedException {
    while (count == 0)
        wait();
    int value = buffer[takeOut];
    takeOut = (takeOut + 1) % numSlots;
    count--;
    notifyAll();
    return value;
}
Scala. Launching the Actors

```scala
object Producers {
  def main(args : Array[String]) {
    val coordinator = new Coordinator;
    val producer = new Producer(coordinator);
    val consumer = new Consumer(coordinator);
    coordinator.init(producer,consumer);
    producer.start;
    coordinator.start;
    consumer.start;
  }
}
```

Key Points

- No “shared state” between actors!
- Consumer and Producer actors communicate via `explicit` message sends/receives
  - Producer sends item!
- No `explicit` synchronization to control order of thread operations, i.e., no `synchronized`, `wait`, etc.
  - In our examples, `receive` causes thread to wait. Implicit ordering of operations
- There is also `asynchronous` message passing

Shared-memory vs. Message-passing

- Compare and contrast shared-memory models with message-passing
- Pros and cons of shared-memory models?
- Pros and cons of message-passing models?

Today's Lecture Outline

- Concurrency... Conclusion
  - Producer Consumer in Java and in Scala
  - Memory consistency models
- 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: `inspected = false; X = 0`
Processor A 
Processor B

```
inspected := true X := 1
xa := X
ib := inspected
```

Processor A and Processor B execute their respective code at about the same time. What values do you expect for `xa` and `ib`?
Memory Consistency Models (Sc. p. 611)

Initially: `inspected = false; X = 0`
Processor A  
Processor B  

`inspected := true`  
`X := 1`  
`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 `xa = 0` and `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  
Thread B  

`inspected := true`  
`X := 1`  
`xa := X`  
`ib := inspected`

Java Memory Model

- Java maintains “sequentially consistency”
- 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

Aside: Transactional Memory (TM)

- Premise: locks are too low-level
- Key idea: programmer specifies atomic blocks. System ensures that an atomic block is executed at once
  - Programmer thinks sequentially: `atomic { i++; }`
    - `atomic`{  
      `r1 = i;`
      `r1 += 1;`
      `i = r1;`
    }  
  - `atomic`{  
      `r2 = i;`
      `r2 += 1;`
      `i = r2;`
    }

Aside: Transactional Memory (TM)

- Uses the optimistic concurrency model
  - Assume no conflicts and execute… Monitor for conflicts at runtime, if conflict, rollback
  - Software transactional memory: a runtime that
    - Detect conflicts
    - Support commits, rollbacks and retries
      - If there are no conflicts, commit “atomic block”
    - Otherwise, roll back and retry later
  - Fundamentally difficult to accomplish

Aside: Issues (TM)?

- What do we do with IO?
- Clearly, a data race that involves two atomic sections is a conflict
  - Strong atomicity
- But what if there is a data race between an atomic section and code outside of an atomic section?
  - Weak atomicity vs. strong atomicity
Aside: Roadblocks (TM)?

- What is the granularity of an atomic section?
  - 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

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

- Following slides (29-38) are based on
  - “The Rise of Dynamic Languages” by Jan Vitek

2 Decades of Dynamic Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Year</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VisualBasic</td>
<td>1991</td>
<td>dyn</td>
</tr>
<tr>
<td>Python</td>
<td>1991</td>
<td>dyn</td>
</tr>
<tr>
<td>Lua</td>
<td>1993</td>
<td>dyn</td>
</tr>
<tr>
<td>R</td>
<td>1993</td>
<td>dyn</td>
</tr>
<tr>
<td>Java</td>
<td>1995</td>
<td>stat+dyn</td>
</tr>
<tr>
<td>JavaScript</td>
<td>1995</td>
<td>dyn</td>
</tr>
<tr>
<td>Ruby</td>
<td>1995</td>
<td>dyn</td>
</tr>
<tr>
<td>PHP</td>
<td>1995</td>
<td>dyn</td>
</tr>
<tr>
<td>C#</td>
<td>2001</td>
<td>stat+dyn</td>
</tr>
<tr>
<td>Scala</td>
<td>2002</td>
<td>stat</td>
</tr>
<tr>
<td>F#</td>
<td>2003</td>
<td>stat</td>
</tr>
<tr>
<td>Clojure</td>
<td>2007</td>
<td>dyn</td>
</tr>
</tbody>
</table>

Last decade or so:

- Go stat ~2009
- Rust stat 2010
- Swift stat 2014
- TypeScript stat+dyn 2015
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();
    }
    ```

- **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;
      // 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

Other Characteristics

- High-level data structures, libraries
- Lightweight syntax
- Delay error reporting, error-oblivious
  - Very difficult to trace and fix errors
- Performance challenged
  - C interpreters ~2-5 times slower than Java
  - Java interpreters ~16-43 times slower than Java

A bit on JavaScript

- 95% of all web sites use JavaScript
- 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

- **Objects** have **fields** (field are also known as properties)
- **Field lookup**
  - `x["f"]`, `x.f`
  - `{ "f" : 7 }["g"]` JavaScript is error-oblivious, in the above example, it returns **undefined**, and continues!
- **Field update**
  - `x["f"] = 2`, `x.f = 2`
  - `{ "f" : 0 }["g"] = 10 yields `{ "f" : 0, "g" : 10 }`

- **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++) {
    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 behavior...

```javascript
with statement
```

See

https://www.destroyallsoftware.com/talks/wat

Taking Stock: Design Choices

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

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Variable Model</th>
<th>Parameter Passing Mechanism</th>
<th>Scoping</th>
<th>Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme</td>
<td>Reference model</td>
<td>By value</td>
<td>Static, nested function definitions</td>
<td>Dynamic, type-safe</td>
</tr>
<tr>
<td>Java</td>
<td>Value model for simple types</td>
<td>By sharing</td>
<td>Static</td>
<td>Static and dynamic, type-safe</td>
</tr>
<tr>
<td>C++</td>
<td>Value model</td>
<td>By value and by reference</td>
<td>Static</td>
<td>Static, type-unsafe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Variable Model</th>
<th>Parameter Passing Mechanism</th>
<th>Scoping</th>
<th>Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell</td>
<td>Reference model</td>
<td>By name (lazy evaluation)</td>
<td>Static, nested function definitions</td>
<td>Static, type-safe</td>
</tr>
<tr>
<td>Scala</td>
<td>Reference model</td>
<td>By value/ sharing and by name</td>
<td>Static, nested function definitions</td>
<td>Static, type-safe</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

```python
m=1; j=3

def outer():
    def middle(k):
        def inner():
            m=4;  # new local m
            print m, j, k;
            inner()
        return m, j, k  # 3 element tuple
    m=2;  # new local m
    return middle(j)  # old (global) j

print outer()
print m, j
```

Scott

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

What is the output of this program?

```
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;
    return middle(j)  # old (global) j
print outer()
print m, j
```

So, what model for variables does Python use?

- Reference model for variables

Datatypes

- Numbers (+, *, **, pow, etc) - immutable
- Collections
  - Sequences
    - Strings are immutable!
    - Lists
  - Tuples are immutable!
- Mappings
  - Dictionaries
- Files
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
  s = 'lumberjack'
  for y in s: print y # iterates over chars
  ```
- Use of iterators defines the “Python style”

Functions

- Two forms of function definition. First class values
  ```python
  def incr(x):
    return x+1 # function incr
  incr
  # list of 2 functions
  incrs = [lambda x: x+1, lambda x: x+2]
  ```
- Polymorphism
  ```python
  def intersect(seq1, seq2):
    res = []
    for x in seq1: # iterates over elements in seq1
      if x in seq2: # checks if element is in seq2
        res.append(x) # if so, adds element to list
    return res
  print intersect([1,2,3],[2,4,6]) # [2]
  print intersect((1,2,3),(4,3,5)) # (3, 5)
  print intersect((1,2,3),(1,2,3)) # [1, 2, 3]
  print intersect(('a','b','c'),'a') # ('a')
  print intersect({'a':1,'b':2,'c':3},{'a':1,'b':2,'c':3}) # {'a':1}
  # clearly the intersection is on the keys, not the values!
  ```

Functions

- What is the parameter passing mechanism in Python?
- 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?