Concurrency

Two or more sequences of events occur in parallel

- Multiprogramming
  - A single computer runs several programs at the same time
  - Each program proceeds sequentially
  - Actions of one program may occur between two steps of another

- Multiprocessors
  - Two or more processors may be connected
  - Programs on one processor communicate with programs on another
  - Actions may happen simultaneously

Process: sequential program running on a processor

Concurrency increasing: many cores on single chip

- From white papers and web sites of current projects:
  - “Conventional wisdom is now to double the number of cores on a chip with each silicon generation.”
  - “The target should be 1000s of cores per chip, as this hardware is the most efficient in MIPS per watt, MIPS per area of silicon, and MIPS per development dollar.”
  - “To maximize programmer productivity, programming models should be independent of the number of processors.”
  - “To maximize application efficiency, programming models should support a wide range of data types and successful models of parallelism: data-level parallelism, independent task parallelism, and instruction-level parallelism.”

see VIEW and RAMP projects (Berkeley, Stanford, MIT, CMU, UW, UT Austin, processor companies, ...)

The promise of concurrency

- Speed
  - If a task takes time $t$ on one processor, shouldn’t it take time $t/n$ on $n$ processors?

- Availability
  - If one process is busy, another may be ready to help

- Distribution
  - Processors in different locations can collaborate to solve a problem or work together

- Humans do it so why can’t computers?
  - Vision, cognition appear to be highly parallel activities

Challenges

- Concurrent programs are harder to get right
  - Folklore: Need at least an order of magnitude in speedup for concurrent prog to be worth the effort

- Some problems are inherently sequential
  - Theory – circuit evaluation is P-complete
  - Practice – many problems need coordination and communication among sub-problems

- Specific issues
  - Communication – send or receive information
  - Synchronization – wait for another process to act
  - Atomicity – do not stop in the middle and leave a mess
Basic question for this course

- How can programming languages make concurrent and distributed programming easier?

What could languages provide?

- Example high-level constructs
  - Thread as the value of an expression
  - Pass threads to functions
  - Create threads at the result of function call
- Communication abstractions
  - Synchronous communication
  - Buffered asynchronous channels that preserve msg order
- Concurrency control
  - Mutual exclusion
  - Most concurrent languages provide some form of locking
  - Atomicity is more abstract, less commonly provided

Basic issue: race conditions

- Sample action
  
  ```
  procedure sign_up(person)
  begin
    number := number + 1;
    list[number] := person;
  end;
  ```

- Problem with parallel execution

  ```
  sign_up(fred) || sign_up(bill);
  ```

Resolving conflict between processes

- Critical section
  - Two processes may access shared resource
  - Inconsistent behavior if two actions are interleaved
  - Allow only one process in critical section
- Deadlock
  - Process may hold some locks while awaiting others
  - Deadlock occurs when no process can proceed

Locks and Waiting

- Initialize concurrency control

  ```
  Thread 1:
  <wait>
  sign_up(fred); // critical section
  <signal>

  Thread 2:
  <wait>
  sign_up(bill); // critical section
  <signal>
  ```

  Need atomic operations to implement wait

Mutual exclusion primitives

- Atomic test-and-set
  - Instruction atomically reads and writes some location
  - Common hardware instruction
  - Combine with busy-waiting loop to implement mutex
- Semaphore
  - Avoid busy-waiting loop
  - Keep queue of waiting processes
  - Scheduler has access to semaphore; process sleeps
  - Disable interrupts during semaphore operations
    - OK since operations are short
State of the art

- Concurrent programming is difficult
  - Race conditions, deadlock are pervasive
- Languages should be able to help
  - Capture useful paradigms, patterns, abstractions
- Other tools are needed
  - Testing is difficult for multi-threaded programs
  - Many race-condition detectors being built today
    - Static detection: conservative, may be too restrictive
    - Run-time detection: may be more practical for now

Concurrent language examples

- Language Examples
  - Cobegin/coend
  - Multilisp futures
  - Actors (C. Hewitt)
  - Concurrent ML
  - Java
- Some features to compare
  - Thread creation
  - Communication
  - Concurrency control (synchronization and locking)

Cobegin/coend

- Limited concurrency primitive
- Example

```plaintext
x := 0;
cobegin
    begin x := 1; x := x+1 end;
    begin x := 2; x := x+1 end;
coend;
print(x);
```

Properties of cobegin/coend

- Advantages
  - Create concurrent processes
  - Communication: shared variables
- Limitations
  - Mutual exclusion: none
  - Atomicity: none
  - Number of processes is fixed by program structure
  - Cannot abort processes
    - All must complete before parent process can go on

History: Concurrent Pascal, P. Brinch Hansen, Caltech, 1970s

Multilisp future

- Example

```lisp
(define (split x) ...)
(define (merge x y) ... (car x) ...)
(define (mergesort x)
  (let ((y, z) (split x))
    (merge (mergesort y) (mergesort z))))
```

- How to rewrite as concurrent algorithm?

Some general approaches

- Explicit concurrency
  - Fork or create threads explicitly
  - Explicit communication between threads
    - Producer computes useful value
    - Consumer requests or waits for producer
- Implicit concurrency
  - Rely on compiler to identify potential parallelism
  - Problems
    - Instruction-level and loop-level parallelism can be inferred, but inferring larger "subroutine"-level parallelism has had less success
Middle Ground: Futures

- Use future annotation [Halstead 85]
  - (future e) indicates e may run concurrently with parent
- Benefits
  - Notationally lightweight
    - Sequential algorithm still expressed in code
  - Concurrency determined by the run-time system
    - Can be based on system resources
  - Simple coordination between threads

Where to annotate?

- (define (split x) ...)
- (define (merge x y) ... (car x) ...)
- (define (mergesort x)
  (let ((y,z) (split x))
   (merge (mergesort y) (mergesort z))))

- No - result is used immediately in following call

Where to annotate?

- (define (split x) ...)
- (define (merge x y) ... (car x) ...)
- (define (mergesort x)
  (let ((y,z) (split x))
   (merge (mergesort y) (mergesort z)))

- Yes - recursive calls can operate in parallel

Multilisp Merge Sort

- (define (split x) ...)
- (define (merge x y) ... (car x) ...)
- (define (mergesort x)
  (let ((y,z) (split x))
   (merge (future (mergesort y)
        (future (mergesort z)))))

Basic Implementation Approach

- (future e)
  - fork a new thread T to evaluate e
  - return a proxy p to the parent
    - called a future or promise
- Producer
  - Thread T stores result of e into proxy p
- Consumer
  - Run-time system extracts result from p when accessed by the parent
    - Called a touch or claim

Implementing Touches

- (define (merge x y) ... (car x) ...)
- Futurized implementation of (car x)
  (if (pair? (touch x))
    (get first elem of x)
    (error))

- Where (touch x) is
  (if (future? x) (get x) x)

- Blocks until result has been computed
Optimization I

- Forking a thread per future could be expensive and without advantage
  - Particularly if not many CPUs
- Idea: only use as many threads as there are processors [Mohr et al 91]
  - At a future call, use idle thread, if any
  - Otherwise, continue using current thread
    - Save continuation on a separate queue
  - When a thread would block, save the current continuation and grab one from the queue

Optimization II

- Once a future computation completes, its result is immutable
  - Proxy and further touches redundant
- Thus
  - Use garbage collector to throw away the proxy and replace with the result [Halstead 85]
  - Avoid touching at all if static analysis can prove it’s unnecessary [Flanagan & Felleisen 95]

Actors

- Each actor (object) has a script
- In response to input, actor may atomically
  - create new actors
  - initiate communication
  - change internal state
- Communication is
  - Buffered, so no message is lost
  - Guaranteed to arrive, but not in sending order
    - Order-preserving communication is harder to implement
    - Programmer can build ordered primitive from unordered
    - Inefficient to have ordered communication when not needed

Actor-Oriented Programs

Object orientation:

```
| call | return |
+------+--------|
| data | methods |
```

What flows through an object is sequential control

Actor orientation:

```
<table>
<thead>
<tr>
<th>input data</th>
<th>output data</th>
</tr>
</thead>
<tbody>
<tr>
<td>actor name</td>
<td>data (state)</td>
</tr>
<tr>
<td>parameters</td>
<td>posts</td>
</tr>
</tbody>
</table>
```

What flows through an object is streams of data

Example

```
1, 4, 7

Insert 2

1, 2, 4, 7
```

Actor program

```
Stack node

a stack_node with acquaintances content and link
if operation requested is a pop and content != nil then
  become forwarder to link
  send content to customer
if operation requested is push(new_content) then
  let P=new stack_node with current acquaintances
  (a clone)
  become stack_node with acquaintances new_content and P
```

Hard to read but it does the “obvious” thing, except that the concept of forwarder is unusual...
Forwarder

- Stack before pop
  - 3 - 4 - 5 nil

- Stack after pop
  - forwarder - 4 - 5 nil

- Node "disappears" by becoming a forwarder node. The system manages forwarded nodes in a way that makes them invisible to the program. (Exact mechanism doesn't really matter since we're not that interested in Actors.)

Concurrency

- Several actors may operate concurrently

- Concurrency not controlled explicitly by program
  - Messages sent by one actor can be received and processed by others sequentially or concurrently

Pros and Cons of Actor model

- High-level programming language
  - Communication by messages
  - Mutual exclusion: if two msgs sent, actor reacts atomically to first one received before seeing second
  - Concurrency is implicit; no explicit fork or wait

- Possibly too abstract for some situations?
  - How do you fork several processes to do speculative computation, then kill them all when one succeeds?
    - Seems to require many msgs to actor that tells all others whether to proceed; this "coordinator" becomes a bottleneck

Concurrent ML [Reppy, Gansner, ...]

- Threads
  - New type of entity

- Communication
  - Synchronous channels

- Synchronization
  - Channels
  - Events
  - Atomicity
    - No specific language support

Pre-Java Concept: Monitor

- Synchronized access to private data
- Combines
  - private data
  - set of procedures (methods)
  - synchronization policy
    - At most one process may execute a monitor procedure at a time; this process is said to be in the monitor
    - If one process is in the monitor, any other process that calls a monitor procedure will be delayed

- Modern terminology: synchronized object

Java Concurrency

- Threads
  - Create process by creating thread object
- Communication
  - Shared variables
  - Method calls

- Mutual exclusion and synchronization
  - Every object has a lock (inherited from class Object)
    - synchronized methods and blocks
    - Synchronization operations (inherited from class Object)
      - wait : pause current thread until another thread calls notify
      - notify : wake up waiting threads
Java Threads

- **Thread**
  - Set of instructions to be executed one at a time, in a specified order

- **Java thread objects**
  - Object of class Thread
  - Methods inherited from Thread:
    - `start`: method called to spawn a new thread of control; causes VM to call run method
    - `suspend`: freeze execution
    - `interrupt`: freeze execution and throw exception to thread
    - `stop`: forcibly cause thread to halt

---

Java Thread States

- **Non-Existing**
  - create thread object
  - garbage collected

- **New**
  - destroy

- **Executable**
  - start
  - run method
  - exit

- **Blocked**
  - wait, join
  - notify, notifyAll

- **Dead**
  - destroy
  - thread termination

---

Problem with language specification

- **Java Lang Spec allows access to partial objects**
  - class `Broken`
    - private long `x`
    - `Broken() { new Thread() { public void run() { x = -1; } }.start(); x = 0; }`
  
  Thread created within constructor can access the object not fully constructed

---

Interaction between threads

- **Shared variables**
  - Two threads may assign/read the same variable
  - Programmer responsibility
    - Avoid race conditions by explicit synchronization !!

- **Method calls**
  - Two threads may call methods on the same object

- **Synchronization primitives**
  - Each object has internal lock, inherited from Object
  - Synchronization primitives based on object locking

---

Synchronization

- **Provides mutual exclusion**
  - Two threads may have access to some object
  - If one calls a synchronized method, this locks object
  - If the other calls a synchronized method on same object, this thread blocks until object is unlocked

---

Synchronized methods

- **Marked by keyword**
  - `public synchronized void commitTransaction(...) { ... }
  - Provides mutual exclusion
    - At most one synchronized method can be active
    - Unsynchronized methods can still be called
    - Programmer must be careful
  - Not part of method signature
    - sync method equivalent to unsync method with body consisting of a synchronized block
    - subclass may replace a synchronized method with unsynchronized method
Example

```java
class LinkedCell {
    // Lisp-style cons cell containing
    protected double value; // value and link to next cell
    protected final LinkedCell next;

    public LinkedCell (double v, LinkedCell t) {
        value = v; next = t;
    }

    public synchronized double getValue() {
        return value;
    }

    public synchronized void setValue(double v) {
        value = v; // assignment not atomic
    }

    public LinkedCell next() { // no synch needed
        return next;
    }
}
```

Join, another form of synchronization

- Wait for thread to terminate
  ```java
class Future extends Thread {
    private int result;
    public void run() {
        result = f(...);
    }
    public int getResult() {
        return result;
    }
}
```

```java
Future t = new Future;
t.start(); // start new thread
...
t.join(); x = t.getResult(); // wait and get result
```

Producer-Consumer?

- Method call is synchronous
- How do we do this in Java?

Solution to producer-consumer

- Cannot be solved with locks alone
  - Use wait and notify methods of Object
- Basic idea
  - Consumer must wait until something is in the buffer
  - Producer must inform waiting consumers when item available
- More details
  - Consumer waits
    - While waiting, must sleep
    - This is accomplished with the wait method
    - Need condition recheck loop
  - Producer notifies
    - Must wake up at least one consumer
    - This is accomplished with the notify method

```java
Stack<T>: produce, consume methods

```java
public synchronized void produce (T object) {
    stack.add(object); notify();
}

public synchronized T consume () {
    while (stack.isEmpty()) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    int lastElement = stack.size() - 1;
    T object = stack.get(lastElement);
    stack.remove(lastElement);
    return object;
}
```

See: http://www1.coe.neu.edu/~jsmith/tutorial.html (also cartoon)
Concurrent garbage collector

- How much concurrency?
  - Need to stop thread while mark and sweep
  - Other GC: may not need to stop all program threads
- Problem
  - Program thread may change objects during collection
- Solution
  - Prevent read/write to memory area
  - Details are subtle; generational, copying GC
    - Modern GC distinguishes short-lived from long-lived objects
    - Copying allows read to old area if writes are blocked ...
    - Relatively efficient methods for read barrier, write barrier

Limitations of Java 1.4 primitives

- No way to back off from an attempt to acquire a lock
  - Cannot give up after waiting for a specified period of time
  - Cannot cancel a lock attempt after an interrupt
- No way to alter the semantics of a lock
  - Reentrancy, read versus write protection, fairness, ...
- No access control for synchronization
  - Any method can perform synchronized(obj) for any object
- Synchronization is done within methods and blocks
  - Limited to block-structured locking
  - Cannot acquire a lock in one method and release it in another

Continue next time ...

Condition rechecks

- Want to wait until condition is true
  public synchronized void lock() throws InterruptedException {
    if (isLocked) wait();
    isLocked = true;
  }
  public synchronized void unlock() {
    isLocked = false;
    notify();
  }
- But need loop since another process may run
  public synchronized void lock() throws InterruptedException {
    while (isLocked) wait();
    isLocked = true;
  }

See http://java.sun.com/developer/technicalArticles/J2SE/concurrency/