Concurrency

CS 456

PURDUE UNIVERSITY

Concurrency in the real world...

- Concurrency can occur at the hardware, operating system and language level
- Categories of Concurrency:
  - *Physical concurrency* - Multiple independent processors (multiple threads of control)
  - *Logical concurrency* - The appearance of physical concurrency is presented by time-sharing one processor (software can be designed as if there were multiple threads of control)
- Concurrency is needed when
  - an application must remain responsive while performing blocking operations, for instance a web server dealing with multiple connections
  - performance must exceed what can be achieved on a single processor
- As the number of cores is increasing faster than clock speed, it seems impossible to ignore concurrency.

Concurrency in the real world...

- Concurrency is a problem
  - Windows 2000: Concurrency errors most common defects among detectable errors
  - Windows 2000: Incorrect synchronization and protocol errors most common coding errors
  - Windows 2003: Synchronization errors second only to buffer overruns
  - Race conditions create security vulnerabilities
  - Concurrent programs are hard to test because they are non-deterministic

  Bugs are hard to reproduce because there are exponentially many possible interleavings that often only manifest on deployed configurations.

Two General Categories of Tasks

- Heavyweight tasks execute in their own address space and have their own run-time stacks
- Lightweight tasks all run in the same address space and use the same run-time stack
- A task is disjoint if it does not communicate with or affect the execution of any other task in the program in any way

Task Synchronization

- A mechanism that controls the order in which tasks execute
- Two kinds of synchronization:
- Cooperation:
  - Task A must wait for task B to complete some specific activity before task A can continue its execution, e.g., the producer-consumer problem
  - In Java: wait/notify
- Competition:
  - Two or more tasks must use some resource that cannot be simultaneously used, e.g., a shared counter
  - Competition is usually provided by mutually exclusive access (approaches are discussed later)
  - In Java: synchronized

Scheduler

- Providing synchronization requires a mechanism for delaying task execution
- Task execution control is maintained by a program called the scheduler, which maps task execution onto available processors
- Task execution states:
  - New - created but not yet started
  - Runnable or ready - ready to run but not currently running (no available processor)
  - Running
  - Blocked - has been running, but cannot now continue (usually waiting for some event to occur)
  - Dead - no longer active in any sense

Liveness and Deadlock

- Liveness is a characteristic that a program unit may or may not have
  - In sequential code, it means the unit will eventually complete its execution
  - In a concurrent environment, a task can easily lose its liveness
  - If all tasks in a concurrent environment lose their liveness, it is called deadlock

Data Races

- A *race condition* occurs if two threads access a shared variable at the same time and at least one of the accesses is a write

- Consider the following program when multiple threads are call `deposit()` in parallel:

```
class Account {
  private int balance;

  void deposit(int n) {
     int j = bal;
     bal = j + n;
  }
}
```

Lock-based Synchronization

- Monitors, or locks, must be used to protect every access to a shared location

```
a = x.a; b = x.b;
||
x.a = 1; x.b = 2;
```

- Locks must be held before every read or write to a shared location

```
synchronized(x) {a = x.a; b = x.b;}
||
synchronized(x) {x.a = 1; x.b = 2;}
```

- When multiple locks are used, a lock acquisition protocol must be adhered to by the application to avoid deadlocks

```
synchronized(x){ synchronized(y) {...}}
||
synchronized(y){ synchronized(x) {...}}
```

Puzzlers #1

- How many different values can there be for local variables a and b?

```
a = x.a; b = x.b;    ||    x.a = 1; x.b = 2;
```

Puzzlers #2

- Do the following programs have data races? Are they non-deterministic?

```
long a = x.a;          ||   long b = x.a;

x.i=1;int i=x.i;       ||   x.i=1;int i=x.i;

                  x.a = MAX_LONG
x.a = 0    ||   long a = x.a;  x.a = (a==MAX_LONG)? 0 : a;
```

Puzzlers #3

- Does the following program have data race?

```
interface INC { void inc(); }

class Int implements INC { int i;  void inc() { i++; } }

class SyncInt implements INC {
   INC val;
   SyncInt(INC v){ val = v; }
   synchronized inc() { val.inc(); }
}

...
INC i = new Int();
INC[] arr=new INC[]{new SyncInt(i),new SyncInt(i),new SyncInt(i)};


arr[0].inc();   ||   arr[1].inc();   ||   arr[2].inc();
```

Puzzlers #3

- Does the following program have concurrency problem?

```
class LL {
    int i;
    LL next;
    LL(LL n,int v){next=n;i=v;}
    synchronized swap() {
       synchronized(next) { int t=next.i; next.i= i; i=t; }
    }
}
```

```
LL a=new LL(null,0), b=new LL(a,1); a.next=b;

      a.swap();   ||  b.swap();
```

Puzzlers #3

- A fix?

```
class LL {
    static int K;
    private final int id = K++;
    int i;
    LL next;
    LL(LL n,int v){next=n;i=v;}
    private void _swap() {int t=next.i; next.i= i; i=t;}
    void swap() {
       if (this.id > next.id)
         synchronized (this) {synchronized(next) { _swap(); }
       else
         synchronized (next) {synchronized(this) { _swap(); }
    }
}
```

Puzzlers #3

- A fix?

```
class LL {
    static int K;
    private final int id;
    int i;LL next;


    LL(LL n,int v){
       next=n;i=v;
       synchronized (LL.class) { id = K++; }
    }

    private void _swap() {int t=next.i; next.i= i; i=t;}
    void swap() {
       if (this.id > next.id)
         synchronized (this) {synchronized(next) { _swap(); }
       else
         synchronized (next) {synchronized(this) { _swap(); }
    }
}
```

Puzzlers #4

- The following idioms occurs frequently in library code. Why is it?

```
final class SyncTree {

   Node left, right;

   private final Object lock = new Object();

   public balance() {

       synchronized (lock) {

           ....
```

Puzzlers #5

- Is this a correct solution to synchronization problems?

```
class Big {
    static final Object Lock = new Object();
}


class LL {
     int i;
     LL next;
     LL(LL n,int v){next=n;i=v;}
     void swap() {
        synchronized(Big.Lock) { int t=next.i; next.i= i; i=t; }
     }
 }


    LL a=new LL(null,0), b=new LL(a,1); a.next=b;

          a.swap();   ||  b.swap();
```

Puzzlers #6

- Does the following program have a data race? A concurrency problem?

```
 class Big { static final Object Lock = new Object(); }


class LL {
     int i;  LL next; LL(LL n,int v){next=n;i=v;}


     void swap() {
       int t = 0;
       synchronized(Big.Lock) { t=next.i; }
       synchronized(Big.Lock) { next.i= i; }
       synchronized(Big.Lock) { i=t; }
     }
 }
```

Lessons

1) To reason about concurrency one must understand all interleaving of operations performed by each thread
2) Not all high-level commands are atomic
3) Aliasing makes it hard to determine which values are shared, thus one may fail to acquire the right lock (or locks)
4) Lock acquisition protocols must be followed to avoid deadlocks
5) Library classes must protect themselves from clients by implementing their own synchronization
6) Oversynchronization is always safe but significantly decreases concurrency (possibly to the point of making it meaningless?)
7) The definition of data race is too low level to catch all concurrency errors

Java Threads

- The concurrent units in Java are methods named `run`
- A run method code can be in concurrent execution with other such methods
- The process in which the run methods execute is called a thread

```
class myThread extends Thread
        public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```

Controlling Thread Execution

- The Thread class has several methods to control the execution of threads
  - The yield is a request from the running thread to voluntarily surrender the processor
  - The sleep method can be used by the caller of the method to block the thread
  - The join method is used to force a method to delay its execution until the run method of another thread has completed its execution

Sleep

```
public void run() {
  while(true){
    try {
      sleep(1000);          // Pause for 1 second or more
    } catch (InterruptedException e) {
       return;             // caused by thread.interrupt()
    }
    ...                     // takes arbitrary time
  }
}
```

Thread Priorities

- A thread's default priority is the same as the thread that create it
  - If main creates a thread, its default priority is NORM_PRIORITY
  - Threads defined two other priority constants, MAX_PRIORITY and MIN_PRIORITY
  - The priority of a thread can be changed with the methods setPriority

Competition Synchronization with

- A method that includes the synchronized modifier disallows any other method from running on the object while it is in execution

…

```
public synchronized void deposit( int i) {…}
public synchronized int fetch() {…}
```

…

- The above two methods are synchronized which prevents them from interfering with each other
- If only a part of a method must be run without interference, it can be synchronized through synchronized statement

```
synchronized (expression) statement
```

Cooperation Synchronization

- Cooperation synchronization in Java is achieved via `wait`, `notify`, and `notifyAll` methods
  - All methods are defined in Object, which is the root class in Java, so all objects inherit them
- The `notify` method is called to tell one waiting thread that the event it was waiting has happened
- The `notifyAll` method awakens all of the threads on the object's wait list

Cooperative synchronization

- The typical scenario is that an application is waiting for some condition
- In a non preemptive system the following would cause a deadlock:

```
while ( condition == false ) {}
... do stuff ...
```

  - with preemption the code is merely inefficient (hogging CPU)
  - the above could be ok if the condition is expected to be mostly true
- This is safer as it lets another thread execute but may still use up CPU unnecessarily if the condition is mostly false

```
while ( condition == false ) Thread.currentThread.yield();
... do stuff ...
```

Cooperative synchronization

- Another approach is to signal the thread with notify()

```
synchronized (lock) { // acquire the lock
   while (condition == false) // check condition
      lock.wait();    // put on wait queue
   condition = false;
}
   ... do stuff...
||

...make condition true...
lock.notify();
```

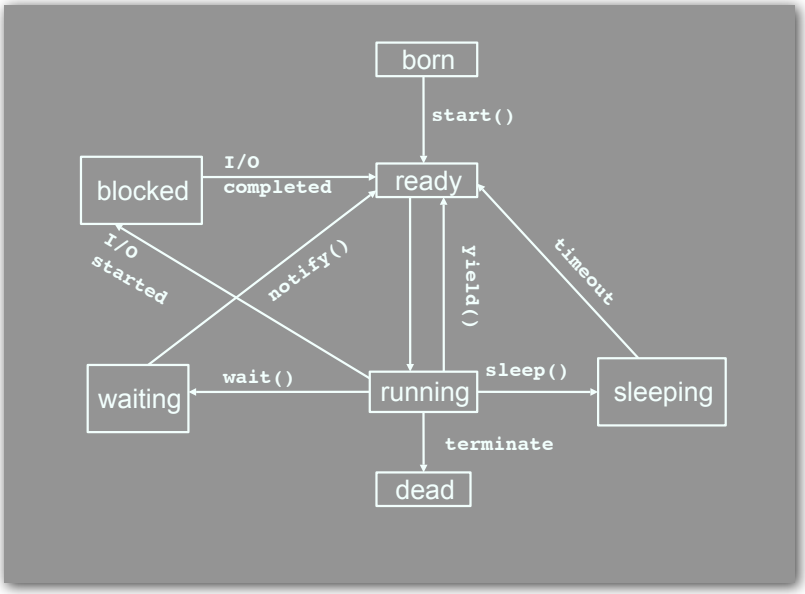
- Calling wait() causes the release of the lock, it is reaquired on wakeup
- `notify()` only wakes up one thread, if there can be more than one consumers use `notifyAll()`

Cooperative synchronization

- Another way to synchronize is with asynchronous signals via Thread.interrupt(), this is mostly cooperative and must be checked via polling. Certain blocking method will throw the interrupted exception

```
synchronized (lock) { // acquire the lock
   while (condition == false) // check condition
     try{
       lock.wait();    // put on wait queue
     } catch (InterruptedException e) {
       ... do something
     }
   condition = false;
}
   ... do stuff...
||

otherThread.interrupt();
```

Thread States



All of this was pretty easy...

... and now on to the hard stuff

Memory Models

- Many programming languages do not deal with concurrency, instead a library provides a set of API calls.
- This is brittle in the presence of optimizing compilers and modern architectures
- Java is the first programming language to specify a *memory model* that must be enforced by the compiler
- The memory model gives semantics to concurrent programs, a must if we want to program multicores
  - C++ is getting one -- it must be good

Motivation: Double Checked Locking

- Double checked locking is an idiom that tries to avoid paying the cost of synchronization when not needed
  - Important for lazy initalization
- The prototypical example:

```
// Single threaded version
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
        helper = new Helper();
    return helper;
    }
  // other functions and members...
  }
```

See Bill Pugh's web page for the details.

Broken Double Checked Locking

- This avoids the cost of synchronization when helper is already initialized:

```
class Foo {
  private Helper helper;
  Helper getHelper() {
    if (helper == null)
      synchronized(this) {
        if (helper == null) helper = new Helper();
      }
    return helper;
  }
```

- Problems:
  - The writes that initialize the Helper object and the write to the helper field can be out of order. A thread invoking getHelper() could see a non-null reference to a helper object, but see the default values for its fields, rather than the values set in the constructor.
  - If the compiler inlines the call to the constructor, then the writes that initialize the object and the write to the helper field can be freely reordered.
  - Even if the compiler does not reorder, on a multiprocessor the processor or memory system may reorder those writes, as perceived by a thread running on another processor.

More on why it's broken

- **A test case showing that it doesn't work** (by Paul Jakubik). When run on a system using the Symantec JIT:

```
singletons[i].reference = new Singleton();
```

to the following (note that the Symantec JIT using a handle-based object allocation system).

```
0206106A   mov         eax,0F97E78h
0206106F   call        01F6B210                 ; allocate space for
                                                ; Singleton, return result in eax
02061074   mov         dword ptr [ebp],eax      ; EBP is &singletons[i].reference
                                                ; store the unconstructed object
here.
02061077   mov         ecx,dword ptr [eax]      ; dereference the handle to
                                                ; get the raw pointer
02061079   mov         dword ptr [ecx],100h     ; Next 4 lines are
0206107F   mov         dword ptr [ecx+4],200h   ; Singleton's inlined constructor
02061086   mov         dword ptr [ecx+8],400h
0206108D   mov         dword ptr [ecx+0Ch],0F84030h
```

As you can see, the assignment to singletons[i].reference is performed before the constructor for Singleton is called. This is completely legal under the existing Java memory model, and also legal in C and C++ (since neither of them have a memory model).

The Java™ Memory Model: the building block of concurrency

Jeremy Manson, Purdue University
William Pugh, Univ. of Maryland
http://www.cs.umd.edu/~pugh/java/memoryModel/

TS-1630

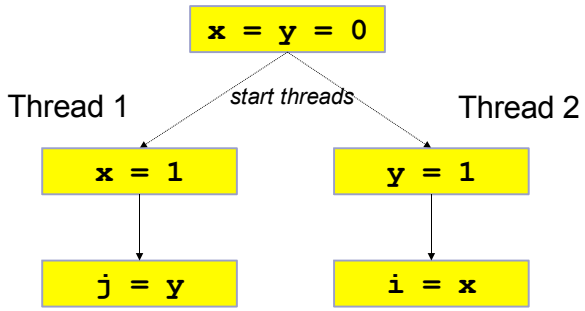
Synchronization is needed for Blocking *and* Visibility

- Synchronization isn't just about mutual exclusion and blocking
- It also regulates when other threads *must* see writes by other threads
  - When writes become visible
- Without synchronization, compiler and processor are allowed to reorder memory accesses in ways that may surprise you
  - And break your code

Don't Try To Be Too Clever

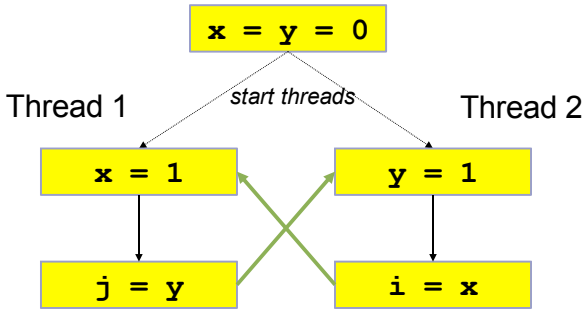
- People worry about the cost of synchronization
  - Try to devise schemes to communicate between threads without using synchronization
    - locks, volatiles, or other concurrency abstractions
- Nearly impossible to do correctly
  - Inter-thread communication without synchronization is not intuitive

Quiz Time



Can this result in i = 0 and j = 0?

Answer: Yes!

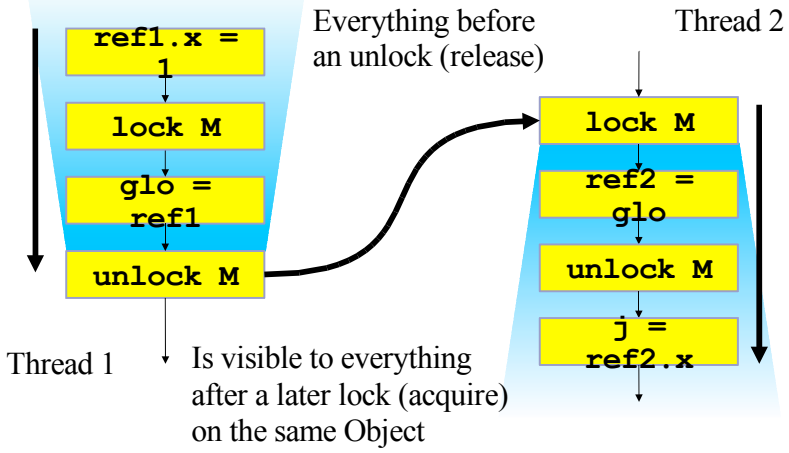


How can i = 0 and j = 0?

How Can This Happen?

- Compiler can reorder statements
  - Or keep values in registers
- Processor can reorder them
- On multi-processor, values not synchronized to global memory
- The memory model is designed to allow aggressive optimization
  - including optimizations no one has implemented yet
- Good for performance
  - bad for your intuition about insufficiently synchronized code

When Are Actions Visible to Other Threads?



Release and Acquire

- All memory accesses before a release
  - are ordered before and visible to
  - any memory accesses after a matching acquire
- Unlocking a monitor/lock is a release
  - that is acquired by any following lock of *that* monitor/lock

Happens-before ordering

- A release and a matching later acquire establish a *happens-before* ordering
- execution order within a thread also establishes a happens-before order
- happens-before order is transitive

Data race

- If there are two accesses to a memory location,
  - at least one of those accesses is a write, and
  - the memory location isn't volatile, then
- the accesses *must* be ordered by happens-before
- Violate this, and you may need a PhD to figure out what your program can do
  - not as bad/unspecified as a buffer overflow in C

Volatile fields

- A field marked as **`volatile`** will be treated specially by the compiler/JIT
- read/writes go directly to memory and are never cached in registers
- volatile long/double are atomic
- volatile operations can't be reordered by the compiler
- The following example will only be guaranteed to work with volatile because this ensure that stop is not stored in a register:

```
class Animator implements Runnable {
  private volatile boolean stop = false;
  public void stop() { stop = true; }
  public void run() {
    while (!stop)
      oneStep();
      try { Thread.sleep(100); } …;
  }
  private void oneStep() { /*...*/ }
}
```

Volatile fields

- Volatile fields induce happens-before edges, similar to locking
- Incrementing a volatile is not atomic
  - if threads threads try to increment a volatile at the same time, an update might get lost
- volatile reads are very cheap; volatile writes cheaper than synchronization
- atomic operations require compare and swap; provided in JSR-166 (concurrency utils)

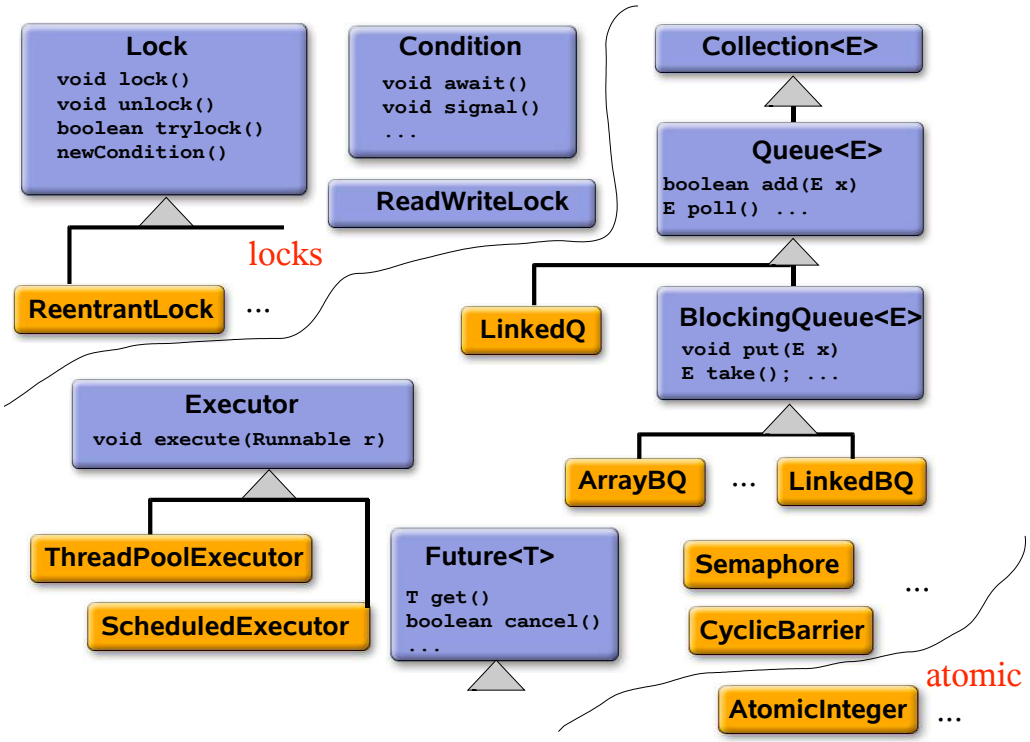
Correct Double Checked Locking

- This avoids the cost of synchronization when helper is already initialized:

```
class Foo {
  private volatile Helper helper;
  Helper getHelper() {
    if (helper == null)
      synchronized(this) {
        if (helper == null) helper = new Helper();
      }
    return helper;
  }
```

Concurrency Utilities and Atomics

- The JSR-166 (designed by Doug Lea at SUNY Oswego) has introduced a rich family of API for concurrent programming



http://gee.cs.oswego.edu

Atomic Variables

- Classes representing scalars supporting

  `boolean compareAndSet(expectedValue, newValue)`

  - Atomically set to `newValue` if currently hold `expectedValue`
  - Also support variant: `weakCompareAndSet`
    - May be faster, but may spuriously fail (as in LL/SC)
- Classes: { *int, long, reference* } X { *value, field, array* } plus boolean value
  - Plus AtomicMarkableReference, AtomicStampedReference
    - (emulated by boxing in J2SE1.5)
- JVMs can use best construct available on a given platform
  - Compare-and-swap, Load-linked/Store-conditional, Locks

Example: AtomicInteger

```
class AtomicInteger {
  AtomicInteger(int initialValue);
  int get();
  void set(int newValue);
  int getAndSet(int newValue);
  boolean compareAndSet(int expected, int newVal);
  boolean weakCompareAndSet(int expected, int newVal);
  //     prefetch                postfetch
  int getAndIncrement();   int incrementAndGet();
  int getAndDecrement();   int decrementAndGet();
  int getAndAadd(int x);   int addAndGet(int x);
}
```

- Integrated with JSR133 memory model semantics for volatile
  - `get` acts as volatile-read
  - `set` acts as volatile-write
  - `compareAndSet` acts as volatile-read and volatile-write
  - `weakCompareAndSet` ordered wrt other accesses to same var

Treiber Stack

```
interface LIFO<E> { void push(E x); E pop(); }

class TreiberStack<E> implements LIFO<E> {
  static class Node<E> {
    volatile Node<E> next;
    final E item;
    Node(E x) { item = x; }
  }

  AtomicReference<Node<E>> head =
        new AtomicReference<Node<E>>();

  public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
      oldHead = head.get();
      newHead.next = oldHead;
    } while (!head.compareAndSet(oldHead, newHead));
  }
```

TreiberStack(2)

```
public E pop() {
   Node<E> oldHead;
   Node<E> newHead;
   do {
     oldHead = head.get();
     if (oldHead == null) return null;
     newHead = oldHead.next;
   } while (!head.compareAndSet(oldHead,newHead));

   return oldHead.item;
 }

}
```


Other Models of Concurrency

Rethinking Concurrency Control

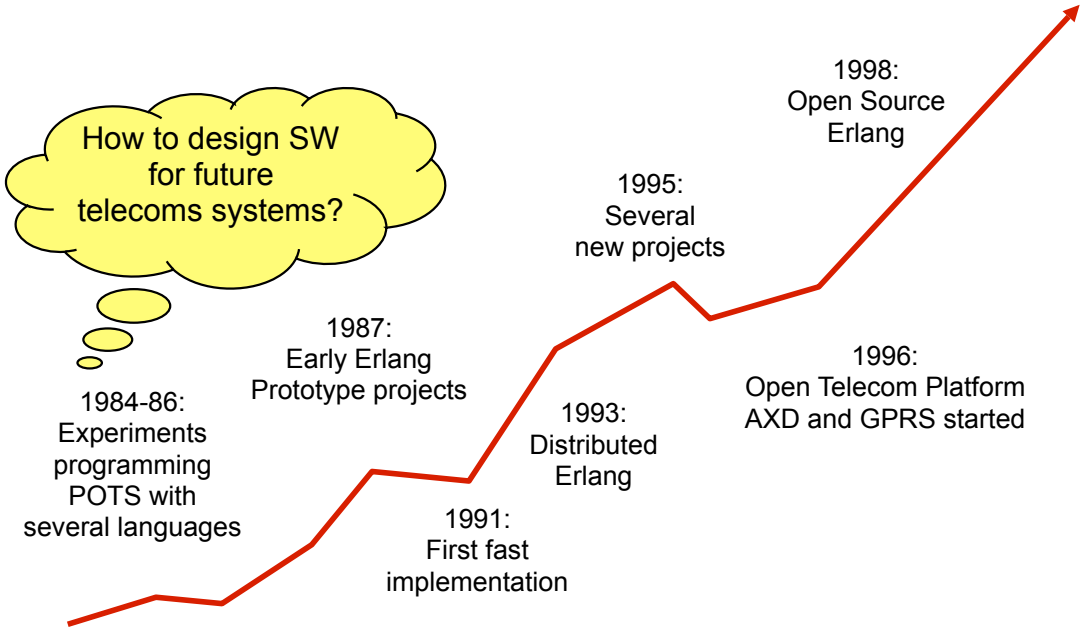
- Most of the problems we have seen so far come from shared memory and lock-based synchronization. Are there alternatives?
- Message passing
  - Erlang -- no mutable state!
  - CML -- little state
- Partitioned address spaces
  - X10
- Atomicity
  - Transactional memory

ERICSSON

Erlang Open Telecom Platform

Ulf Wiger

History of Erlang



Erlang Example

Creating a new process using `spawn`

```
-module(ex3).
-export([activity/3]).

activity(Name,Pos,Size) ->
    ..........
```

```
pid = spawn(ex3,activity,[Joe,75,1024])
```

```
activity(Joe,75,1024)
```

Erlang Example

Processes communicate by asynchronous message passing

```
Pid ! {data,12,13}
```

```
receive
    {start} -> .....
    {stop} -> .....
    {data,X,Y} -> .....
end
```


Erlang Examples

Concurrency - Finite State Machine

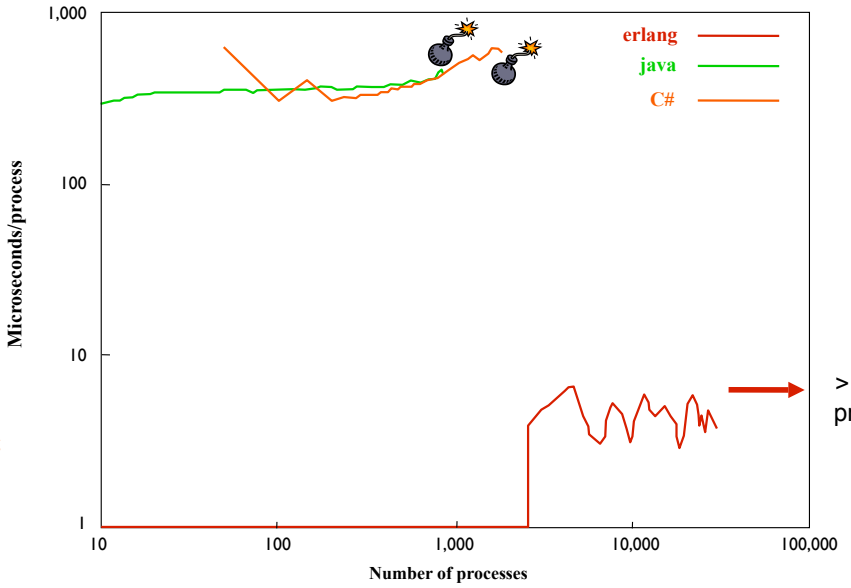
```
ringing_B_side(PidA) ->
    receive
      {lim, offhook} ->
          lim:stop_ringing(),
          PidA ! {hc, {connect, self()}},
          speech(PidA);
      {hc, {cancel, PidA}} ->
          cancel(PidA);
      {lim, {digit, _Digit}} ->
          ringing_B_side(PidA);
      {hc, {request_connection, Pid}} ->
          Pid ! {hc, {reject, self()}},
          ringing_B_side(PidA)
      after 30000 ->
          cancel(PidA)
    end.
```

Selective receive

Asynchronous send

Optional timeout

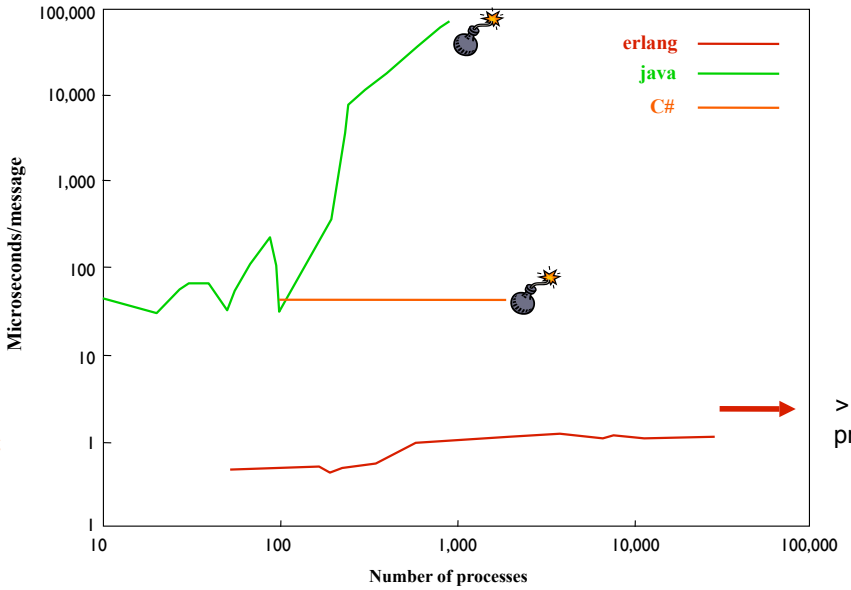
Process creation times (LOG/LOG



Source:
Joe Armstrong
SICS

> 200,000
processes

Message passing times (LOG/LOG



Source:
Joe Armstrong
SICS

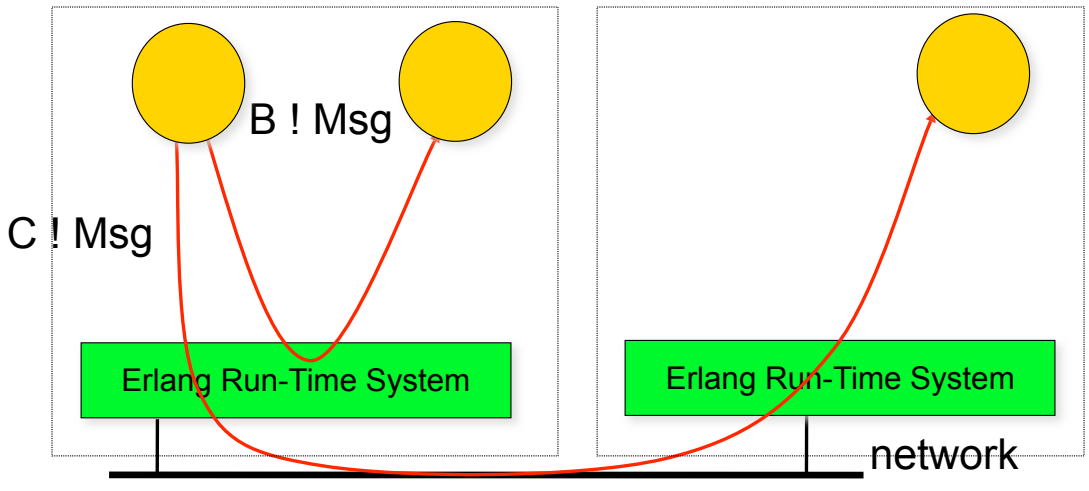
> 200,000
processes

Error-handling -- Language Safety

- No global variables -- fewer side-effects
- No direct memory access -- no pointer errors
- No malloc/free bugs
- Solid concurrency model -- reduces synchronization problems, reduces the state space (simpler programs)
- Fault isolation -- memory-protected lightweight processes
- Built-in error recovery support -- more consistency

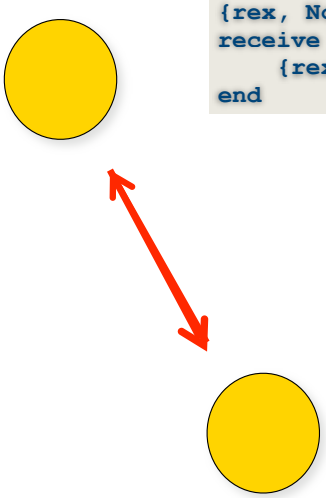
Concurrency & Fault Tolerance were designed into the language from the start!

Transparent Distribution



Message passing between processes in different computer is just as easy as between processes in the same computer

Simple RPC



```
{rex, Node} ! {self(), {apply, M, F, A}},
receive
    {rex, Node, What} -> What
end
```

```
loop() ->
    receive
        {From, {apply, M, F, A}} ->
            Answer = (catch apply(M, F, A)),
            From ! {rex, node(), Answer}
            loop();
        _Other -> loop()
    end.
```


CML

Issues in Concurrent Language Design

John Reppy
AT&T Labs Research

November 1996

Synchronization and communication

The choice of synchronization and communication mechanisms is the most important design choice in a concurrent language.

- Should these be independent or coupled?
- What guarantees should be provided?

Synchronization *(continued ...)*

There are a range of mechanisms found in concurrent languages:

- Shared memory (locks and condition variables)
- Synchronous memory (I-structures and M-structures)
- Asynchronous message passing (buffered channels)
- Synchronous message passing (blocking send)
- RPC (aka extended rendezvous)

Design point:

Shared-memory is a poor programming model

- Although one can write very efficient programs this way, the model does not promote correctness.
- It requires defensive programming (protect your data/code from interference), without compiler support.
- Shared-memory primitives do not fit well with a value-oriented programming style.

Message ordering

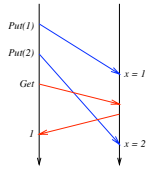
Some recent designs have proposed treating buffered channels as multisets (instead of as queues).

This has implementation advantages in distributed settings, and allows an easy *undo* mechanism for communications.

Design point:

FIFO ordering on messages is good

There are very few interprocess interactions that do not require at least a FIFO ordering on messages.



Transparent distribution

Some people argue that we should implement concurrent languages on distributed systems in a *transparent* fashion — i.e., local and remote communication should look the same.

Design point:

Transparent distribution is problematic

- Remote operations have high latency; local ones do not.
- Transferring large data structures locally is done by pointer copying; remote transfer requires much more work.
- Remote systems/links may fail, but concurrent languages do not provide a model these kinds of failures.

Extended rendezvous

Some languages provide request/reply as the communication mechanism (Ada, Concurrent C).

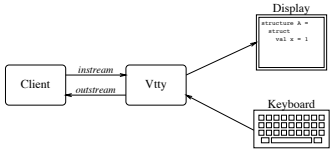
Design point:

Extended rendezvous is too much

- It is easy to implement extended rendezvous on top of message-passing.

Selective communication

In a language with blocking communication operations (e.g., `recv` and blocking `send`), it is useful to choose between a set of blocking operations.



Can solve this by union types and extra threads; in other examples, we may use special protocols.

Design point:

Selective communication is important

- It reduces the number of threads required.
- It promotes modularity, since specialized protocols may not compose.

Design point:

Negative acknowledgements promote modularity

Negative acknowledgements are a mechanism that are unique to CML. They provide a way to implement abortable protocols as abstractions.

Quick CML review

Basic features:

- Explicit threading with preemptive scheduling.
- Threads communicate and synchronize via message passing using a variety of primitives (buffered channels, I-variables, and M-variables).
- Synchronization and communication are supported by the mechanism of *first-class synchronous operations* (called *events*).

Interprocess communication

```
type 'a chan

val channel : unit -> 'a chan
val recv : 'a chan -> 'a
val send : ('a chan * 'a) -> unit
```

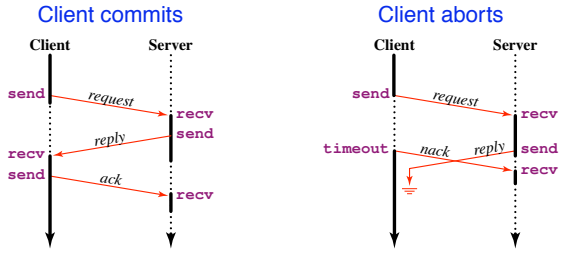
Sending a message is a blocking operation in CML.

Most interactions between processes involve multiple messages.

A process may need to interact with multiple partners (*nondeterministic choice*).

Protocols *(continued ...)*

Here are message sequence diagrams for a *client/server* protocol with acknowledgments.



Events

We use *event* values to package up protocols as abstractions.

An event is an abstraction of a synchronous operation, such as receiving a message or a timeout.

```
type 'a event
```

Base-event constructors create event values for communication primitives:

```
val recvEvt : 'a chan -> 'a event
```

Events allow complicated communication protocols to be implemented as first-class abstractions.

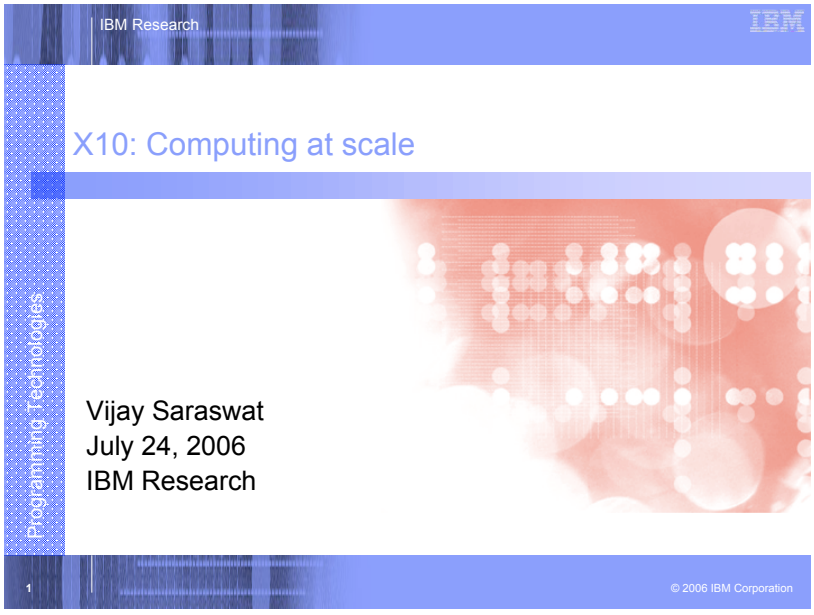
Events *(continued ...)*

CML event operations:

- Event wrappers for post-synchronization actions.
- Event generators for pre-synchronization actions and cancellation.
- Choice for managing multiple communications.
- Synchronization on an event value.

X10

PURDUE UNIVERSITY



Acknowledgments

- **X10 Core Team**
  - **Rajkishore Barik**
  - **Chris Donawa**
  - **Allan Kielstra**
  - **Igor Peshansky**
  - **Christoph von Praun**
  - **Vijay Saraswat**
  - **Vivek Sarkar**
  - **Tong Wen**
- **X10 Tools**
  - **Philippe Charles**
  - **Julian Dolby**
  - **Robert Fuhrer**
  - **Frank Tip**
  - **Mandana Vaziri**
- **Emeritus**
  - **Kemal Ebcioglu**
  - **Christian Grothoff**
- **Research colleagues**
  - **R. Bodik, G. Gao, R. Jagadeesan, J. Palsberg, R. Rabbah, J. Vitek**
  - **Several others at IBM**

Recent Publications

1. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing", P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, V. Sarkar. OOPSLA conference, October 2005.
2. "Concurrent Clustered Programming", V. Saraswat, R. Jagadeesan. CONCUR conference, August 2005.
3. "An Experiment in Measuring the Productivity of Three Parallel Programming Languages", K. Ebcioglu, V. Sarkar, T. El-Ghazawi, J. Urbanic. P-PHEC workshop, February 2006.
4. "X10: an Experimental Language for High Productivity Programming of Scalable Systems", K. Ebcioglu, V. Sarkar, V. Saraswat. P-PHEC workshop, February 2005.

Upcoming tutorials

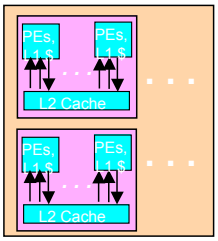
- PACT 2006, OOPSLA 2006

A new Era of Mainstream Parallel Processing

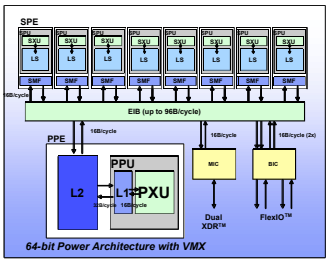
The Challenge:

Parallelism scaling replaces frequency scaling as foundation for increased performance ➔ Profound impact on future software

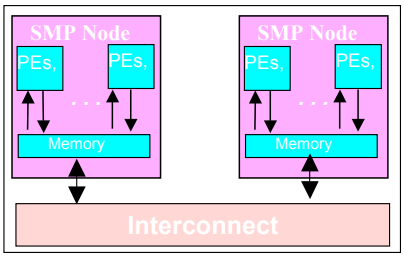
Multi-core chips



Heterogeneous Parallelism



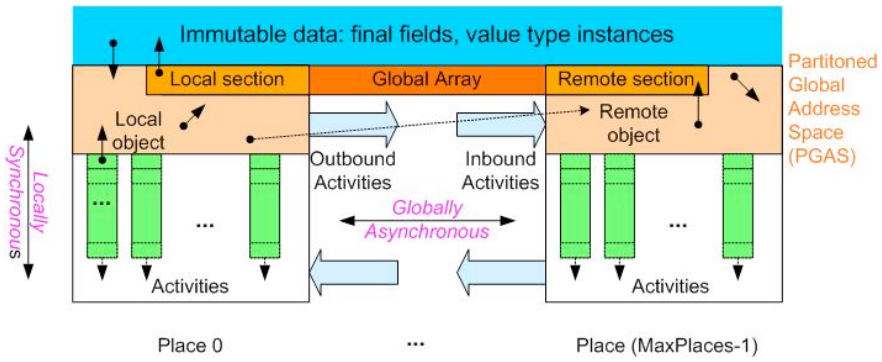
Cluster Parallelism



Our response:

Use X10 as a new language for parallel hardware that builds on existing tools, compilers, runtimes, virtual machines and libraries

The X10 Programming Model



Place = collection of resident activities & objects

Storage classes
- **Immutable Data**
- **PGAS**
  - Local Heap
  - Remote Heap
- **Activity Local**

Locality Rule
Any access to a mutable datum must be performed by a local activity ➔ remote data accesses can be performed by creating remote activities

Ordering Constraints (Memory Model)
Locally Synchronous:
Guaranteed coherence for local heap ➔ Sequential consistency

Globally Asynchronous:
No ordering of inter-place activities ➔ use explicit synchronization for coherence

Few concepts, done right.

Programming Technologies

X10 v0.41 Cheat Sheet

Stm:

async *[* (***Place***) *]* *[*clocked ***ClockList*** *]* ***Stm***

when (***SimpleExpr***) ***Stm***

finish ***Stm***

next; ***c.resume()*** ***c.drop()***

for(***i : Region***) ***Stm***

foreach (***i : Region***) ***Stm***

ateach (***I : Distribution***) ***Stm***

Expr:

ArrayExpr

ClassModifier : Kind

MethodModifier: atomic

DataType:

ClassName | ***InterfaceName*** | ***ArrayType***

nullable ***DataType***

future ***DataType***

Kind :

value | reference

x10.lang has the following classes (among others)

point, range, region, distribution, clock, array

Some of these are supported by special syntax.

Programming Technologies

Value types : immutable instances

- **Value class**
  - Can only extend value class or x10.lang.Object.
  - Have only final fields
  - Can only be extended by value classes.
  - May contain fields at reference type.
  - May be implemented by reference or copy.
- **Two values are equal if their corresponding fields are equal.**
- **nullable _ provided as a type constructor.**

```
public value complex {
  double im, re;
  public complex(double im,
                 double re) {
    this.im = im; this.re = re;
  }
 public complex add(complex a) {
   return new complex(im+a.im,
                      re+a.re);
} … }
```


async, finish

`async PlaceExpressionSingleListopt Statement`

- async (P) S
  - **Parent activity creates a new child activity at place P, to execute statement S; returns immediately.**
  - **S may reference *final* variables in enclosing blocks.**

```
double[D] A =…;  // Global dist. array
final int k = …;
async ( A.distribution[99] ) {
   // Executed at A[99]'s place
   atomic A[99] = k;
}
```

`Statement ::= finish Statement`

- finish S
  - Execute S, but wait until all (transitively) spawned async's have terminated.
  - Trap all exceptions thrown by spawned activities, throw aggregate exception when all activities terminate.

```
finish ateach (point [i]:A) A[i] = i;
finish async (A.distribution[j]) A[j] = 2;
// All A[i]=I will complete before A[j]=2
```

cf Cilk's spawn, sync

atomic, when

Statement ::= atomic *Statement*
MethodModifier ::= atomic

- **Atomic blocks are**
  - Executed in a single step, conceptually, while other activities are suspended.
- **An atomic block may not**
  - Block
  - Access remote data.
  - Create activities.
  - Contain a conditional block.
- **Essentially, body is a bounded, sequential, non-blocking activity**
  - Hence executing in a single place.

Statement ::= *WhenStatement*
WhenStatement ::= when (*Expression*) *Statement*

- **Conditional atomic blocks**
  - Activity suspends until a state in which guard is true; in that state it executes body atomically.
- **Body has same restructions as unconditional atomic block.**
- `await(e)=def=when(e);`
- **X10 does not assume retry semantics for atomics.**

X10 has only one synchronization construct: conditional atomic block.

Atomic blocks simplify parallel programming

- **No explicit locking**
  - No need to worry about lock management details: What to lock, in what order to lock.
- **No underlocking/overlocking issues.**
- **No need for explicit consistency management**
  - No need to carry mapping between locks and data in your head.
- **System can manage locks and consistency better than user**
- **Enhanced performance scalability**
  - X10 distinguishes intra-place atomics from inter-place atomics.
  - Appropriate hardware design (e.g. conflict detection) can improve performance.
- **Enhanced analyzability**
  - First class programming construct
- **Enhanced debuggability**
  - Easier to understand data races with atomic blocks than with critical sections/synchronization blocks

Bounded buffer

```
class OneBuffer {
    nulable Object datum = null;
    public void send( Object v) {
         when (datum == null) {
             datum = v;
          }
    }

    public Object receive() {
        when (datum != null) {
           Object v = (Object) datum;
           datum = null;
           return v;
        }
    }
}
```


Atomic blocks: Barrier synchronization

ORIGINAL JAVA CODE

Main thread (see spec.jbb.Company): …

```
// Wait for all threads to start.
synchronized (company.initThreadsStateChange) {
     while ( initThreadsCount != threadCount ) {
             try {
             initThreadsStateChange.wait();
             } catch (InterruptedException e) {…}
     }
}

// Tell everybody it's time for warmups.
mode = RAMP_UP;
synchronized (initThreadsCountMonitor) {
     initThreadsCountMonitor.notifyAll();
}  …
```

Worker thread (see spec.jbb.TransactionManager): …

```
synchronized (company.initThreadsCountMonitor) {
     synchronized
     (company.initThreadsStateChange) {
             company.initThreadsCount++;

     company.initThreadsStateChange.notify();
     }
     try {

     company.initThreadsCountMonitor.wait();
     } catch (InterruptedException e) {…}
} …
```

EQUIVALENT CODE WITH ATOMIC SECTIONS

Main thread: …

```
// Wait for all threads to start.
when(company.initThreadsCount==thre
    adCount) {
    mode = RAMP_UP;
    initThreadsCountReached = true;
} …
```

Worker thread: …

```
atomic {
    company.initThreadsCount++;
}

await ( initThreadsCountReached );
    //barrier synch.
…
```

Determinate, dynamic barriers: clocks

- Operations
  clock c = new clock();
  c.resume();
    - Signals completion of work by activity in this clock phase.

  next;
    - Blocks until **all** clocks it is registered on can advance. Implicitly resumes all clocks.

  c.drop();
    - Unregister activity with **c**.

No explicit operation to register a clock.

async(P)clocked(c_1,…,c_n)S
- (Clocked async): activity is registered on the clocks (c_1,…,c_n)

- Static Semantics
  - An activity may operate only on those clocks for which it is live.
  - In **finish S**, **S** may not contain any (top-level) clocked asyncs.
- Dynamic Semantics
  - A clock c can advance only when all its registered activities have executed **c.resume().**

Supports over-sampling, hierarchical nesting.

Programming Technologies

regions, distributions

- Region
  - a (multi-dimensional) set of indices
- Distribution
  - A mapping from indices to places
- High level algebraic operations are provided on regions and distributions

```
region R = 0:100;
region R1 = [0:100, 0:200];
region RInner = [1:99, 1:199];
// a local distribution
dist D1=R-> here;
// a blocked distribution
dist D = block(R);
// union of two distributions
dist D = (0:1) -> P0 || (2:N) -> P1;
dist DBoundary = D – RInner;
```

Based on ZPL

Programming Technologies

arrays

- Arrays may be
  - Multidimensional
  - Distributed
  - Value types
  - Initialized in parallel:

```
int [D] A= new int[D]
    (point [i,j]) {return N*i+j;};
```

- Array section
  - **A[RInner]**
- High level parallel array, reduction and span operators
  - Highly parallel library implementation
  - A-B (array subtraction)
  - A.reduce(intArray.add,0)
  - A.sum()

Ateach, foreach

- ateach (point p : A) S
  - Creates |region(A)| async statements
  - Instance p of statement S is executed at the place where A[p] is located
- foreach (point p : R) S
  - Creates |R| async statements in parallel at current place
- Termination of all activities can be ensured using finish.

ateach (*FormalParam*: *Expression*) *Statement*
foreach (*FormalParam*: *Expression*) *Statement*

```
public boolean run() {
 dist D = dist.factory.block(TABLE_SIZE);
 long[.] table = new long[D] (point [i]) { return i; }
 long[.] RanStarts = new long[distribution.factory.unique()]
    (point [i]) { return starts(i);};
 long[.] SmallTable = new long value[TABLE_SIZE]
   (point [i]){return i*S_TABLE_INIT;};
 finish ateach (point [i] : RanStarts ) {
  long ran = nextRandom(RanStarts[i]);
  for (int count: 1:N_UPDATES_PER_PLACE) {
   int J = f(ran);
   long K = SmallTable[g(ran)];
   async atomic table[J] ^= K;
   ran = nextRandom(ran);
  }}
return table.sum() == EXPECTED_RESULT;
}
```

Transactional Memory

(material by Suresh Jagannathan)

PURDUE UNIVERSITY

Observations

- Mainstream adoption of concurrency and distributed programming abstractions
  - Heavy burden on programmer to balance safety and performance
  - Well-known issues with deadlocks, data races, priority inversion, interaction with external actions, etc.
  - Scalability impacted by the use of mutual-exclusion

    Finer-grained locks require more care to prove correct
- Advent of multi-core processors
  - Each core can support multiple threads
  - Programmability remains an open question:

    How much parallelism can a compiler safely extract?
- Can we simplify concurrent program structure without sacrificing efficiency or scalability?
  - Lock-free data structures and algorithms
  - Software transactions (obstruction-free)

Software Transactions

- Instead of strict synchronization semantics induced by lock-based abstractions,
  - Define a relaxed synchronization model:

    Decouples shared access from synchronization machinery

    Allow concurrent access to shared data provided serialization invariants are not violated.
  - Separate specification of program correctness from implementation of a specific solution

    Define a guarded region of code protected by a specific concurrency control protocol.

    Ideally, applications should be able to overspecify the scope of these regions:

    The burden of how and when tasks can concurrently access shared data within these regions is shifted from the application to the implementation.

Goals

- Safety
  - Race-freedom
  - No priority inversion
  - Guarantee serializable execution
- Improved performance
  - Access to shared data structures can take place concurrently provided there is no violation of serializability

    Imposes weaker constraints on implementations
  - Beneficial impact on scalability
- Software engineering
  - Facilitates abstraction

Approaches

- Lock-based concurrency control:
  - Programmer responsible for correct and efficient placement of locks.
- Transactional Memory:
  - Provide two important properties:

    Atomicity: effects of updates seen all-at-once or not-at-all.

    Isolation: while executing within a shared region, effects of other threads not witnessed.
  - Optimistic transactions: allow threads to execute shared (guarded) regions of code assuming serializability will hold.

    When it fails, abort and retry.
  - Pessimistic transactions: associate locks with all shared data and acquire when accessed, and release at end of transaction.

    Deadlock on lock acquires, requires abort and retry.

Basic Actions

- Start
  - monitor access within the dynamic extent of a transaction region
- Log
  - Record updates within a transaction in case an abort occurs
- Abort
  - Restore global state and retry
- Commit
  - Check serializability invariants

Phases

- Optimistic:
  - Read phase: maintain log recording reads and writes to shared data.
  - Validation phase: compare transaction log with global state:

    Abort if comparision reveals a serializability violation.
  - Commit phase: update shared data to the heap.
- Pessimistic:
  - Read phase: acquire locks on shared reads and writes.

    Log original values to handle aborts.

    Abort if a deadlock exists among multiple transactions that require resources (i.e., locks) held by the other.
  - Commit phase: release held locks.

    Updates always immediately performed to the global heap.
- The two approaches are not necessarily exclusive:
  - Consider pessimistic writes and optimistic reads.

    Allows transactions to eagerly abort on conflicting writes.

Foundational Mechanisms

- Logging –
  - versioning used to redirect transactional accesses
  - versioning to used to restore aborted transaction
- Dependency tracking –
  - discover violations of serializability
  - discover deadlocks on lock access

    Granularity of conflict detect (word vs. object)
- Revocation –
  - undo effects of transactions violating serializability and re-execute them
  - undo effects of deadlock transactions
  - contention management:

    When a transaction aborts, when should it run again?

    How should livelocks be prevented?

    Obstruction-freedom

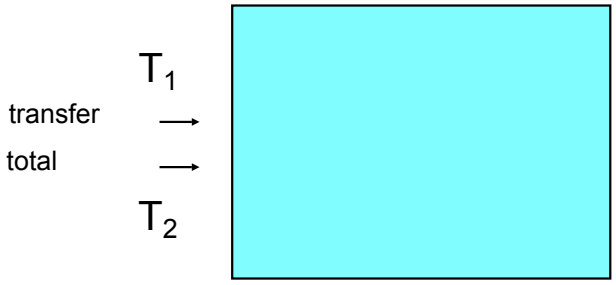
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



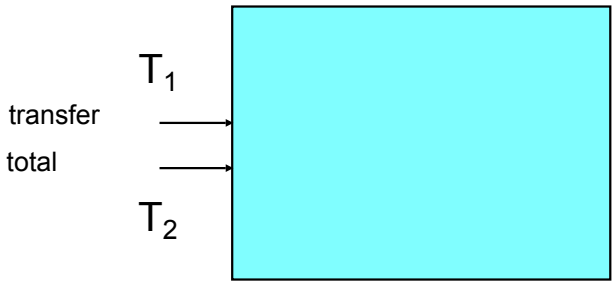
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



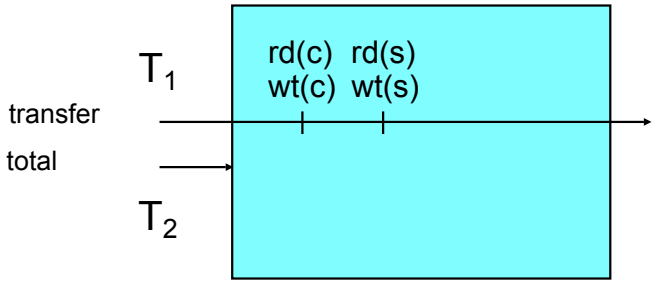
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



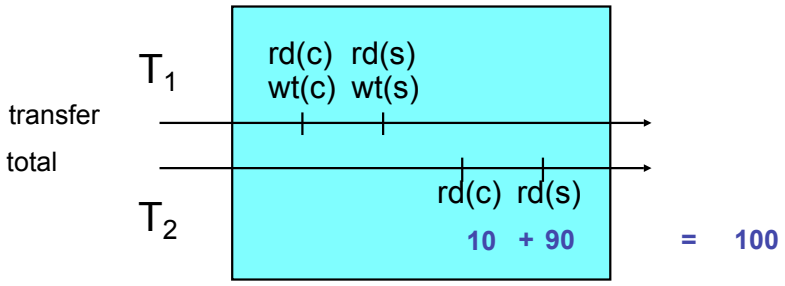
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



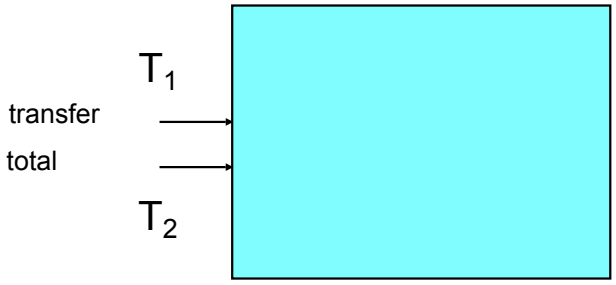
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



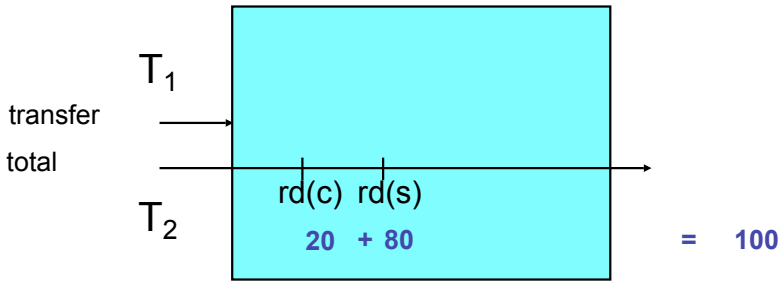
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



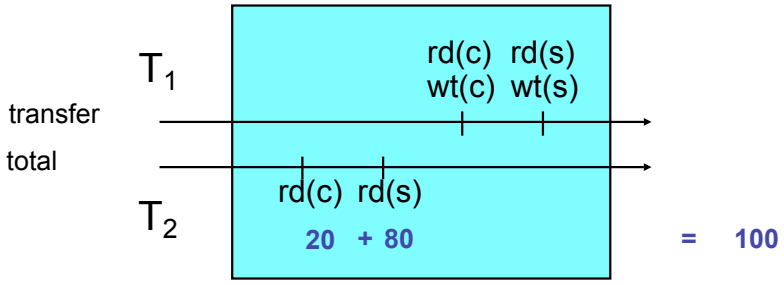
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



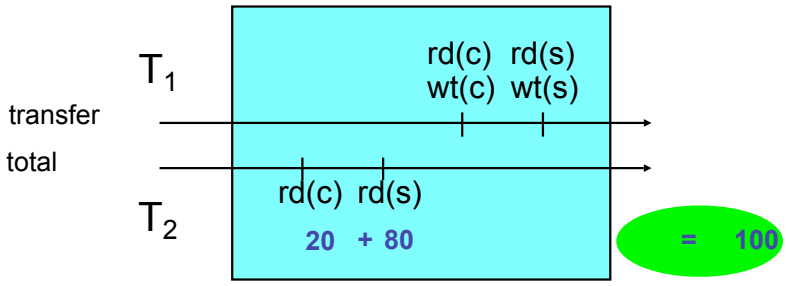
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



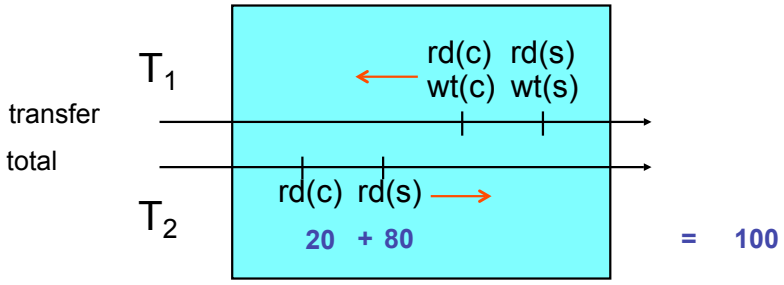
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



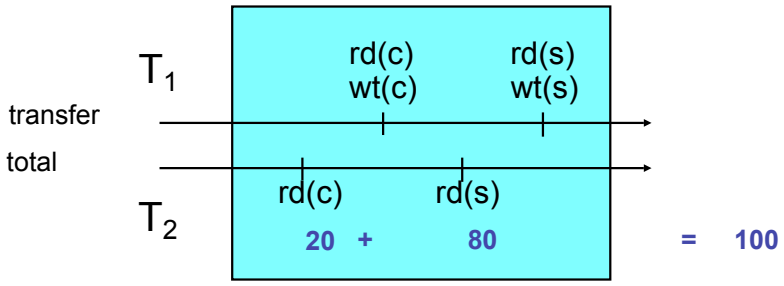
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



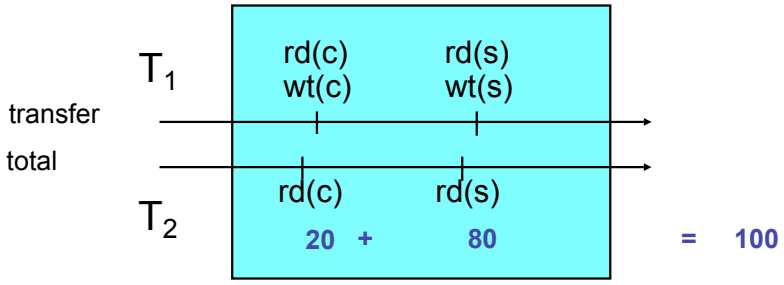
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



Transactional Memory

- In it's simplest form: replace synchronized with atomic

Ensuring Serializability

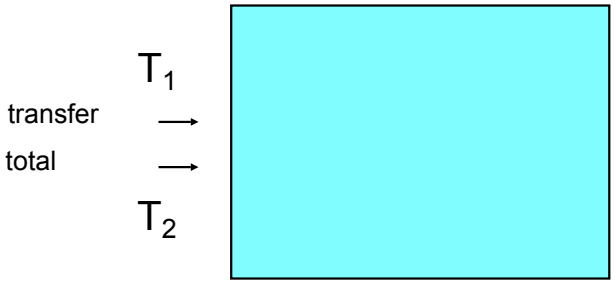
```
// checking   // savings
Account c;    Account s;
```

20 80

atomic

10

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



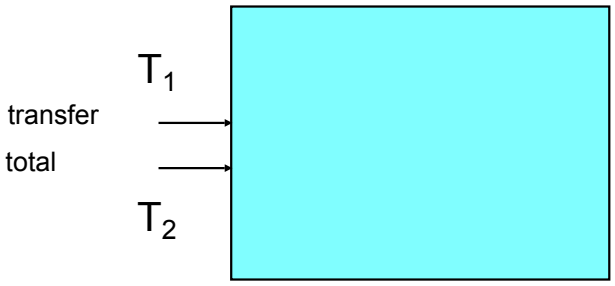
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```

20 80

10

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



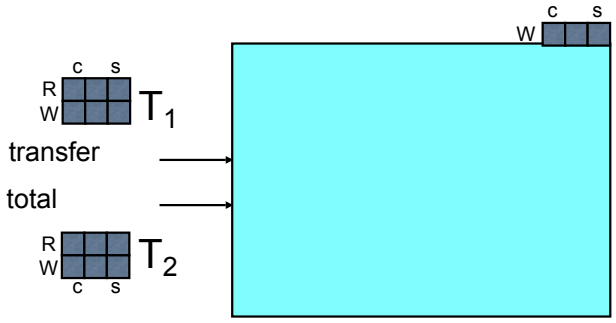
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



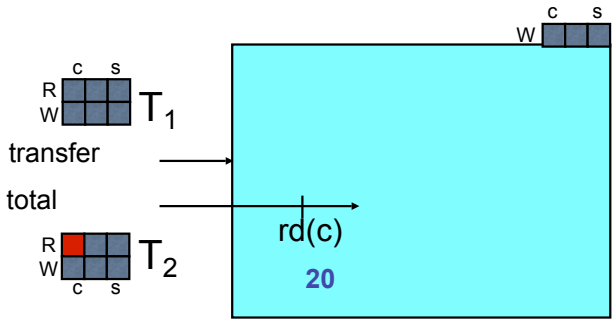
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10

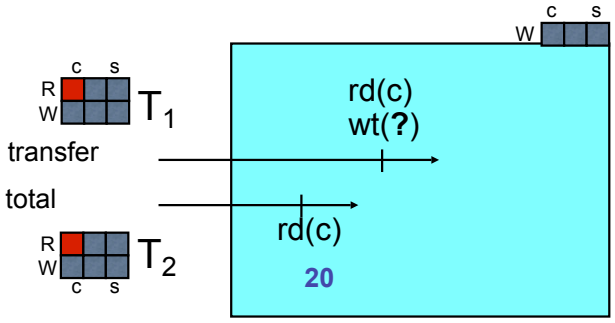


Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

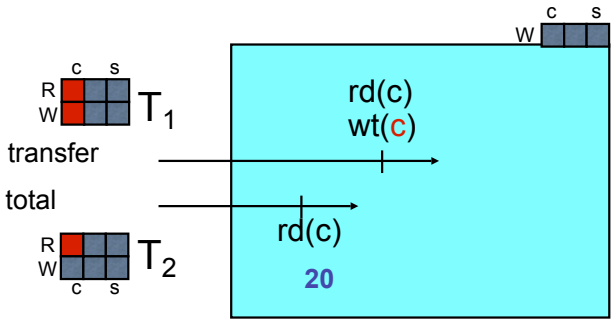


Ensuring Serializability

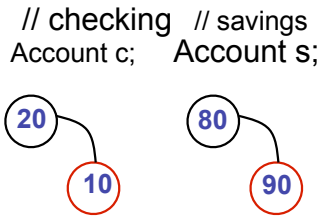
```
// checking   // savings
Account c;    Account s;
```

20 80 10

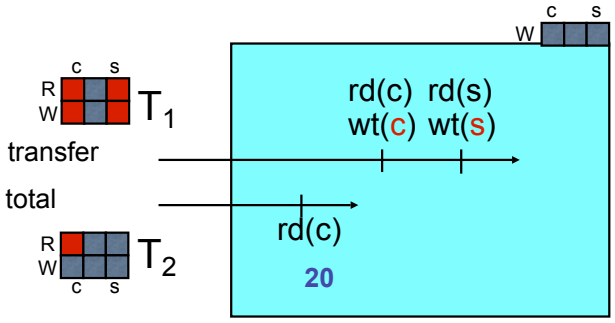
```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



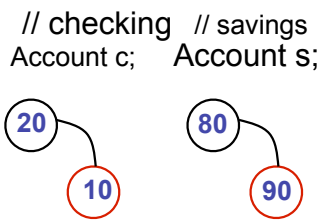
Ensuring Serializability



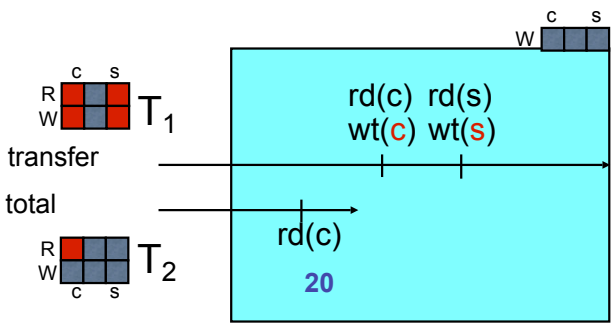

```
10
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



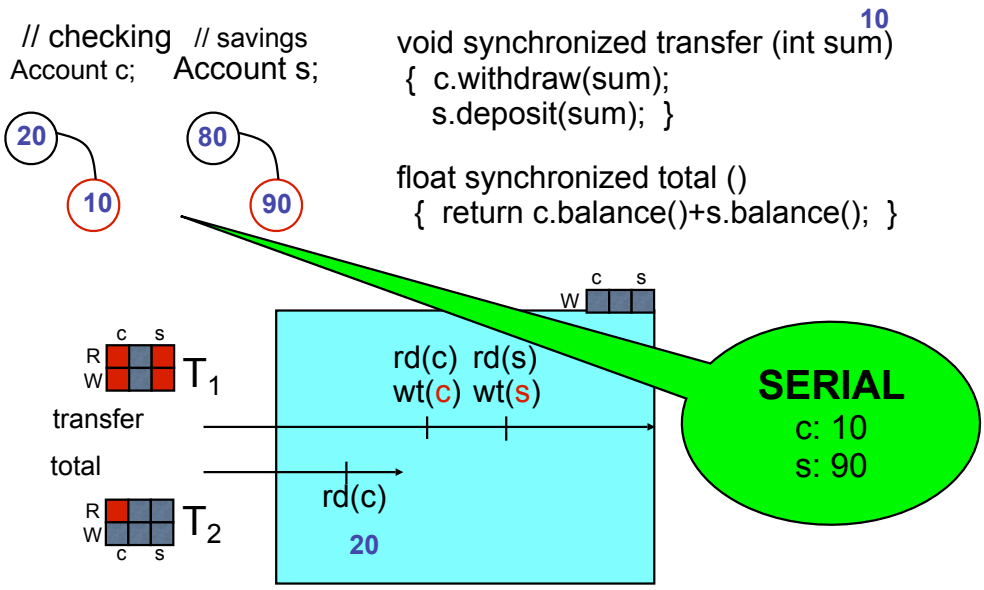
Ensuring Serializability



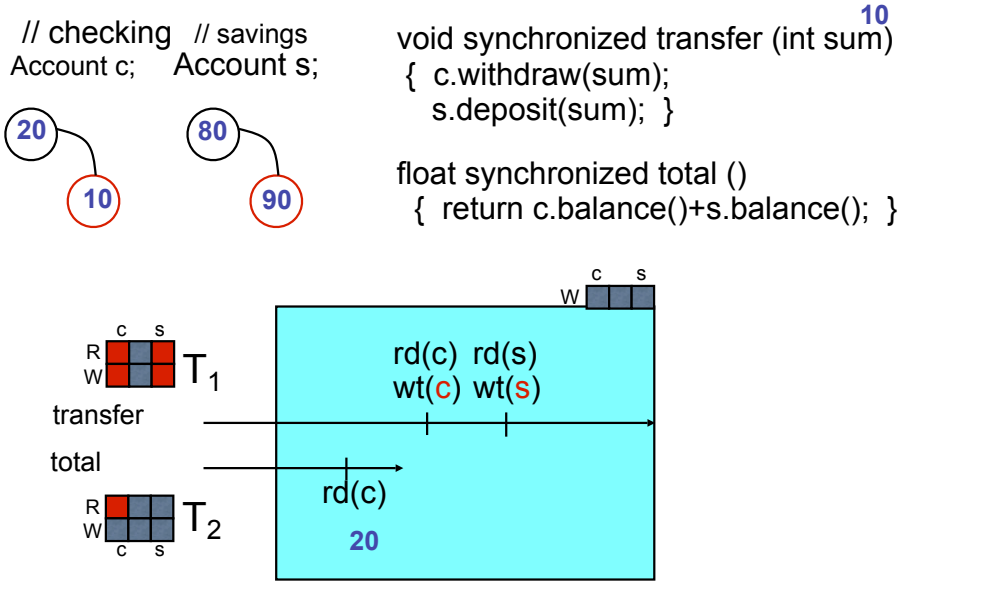

```
10
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



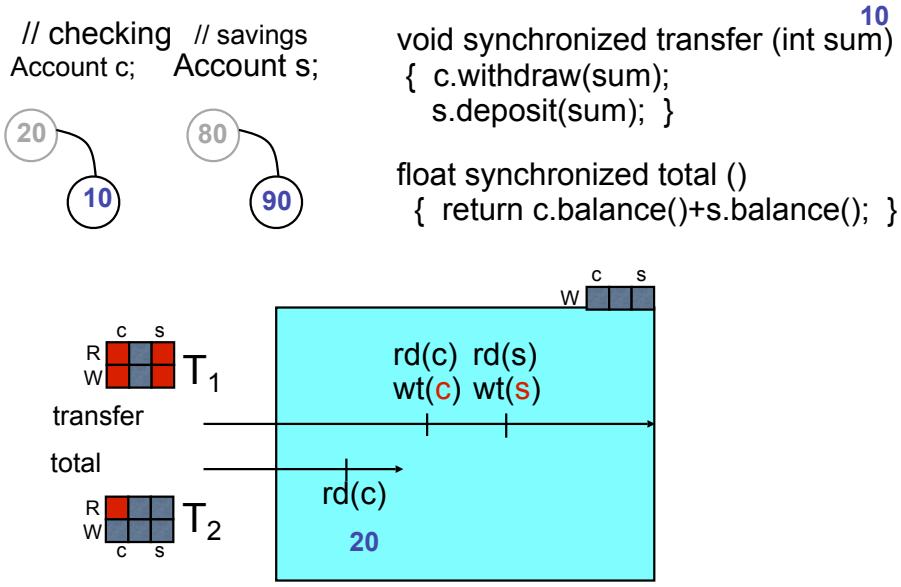
Ensuring Serializability



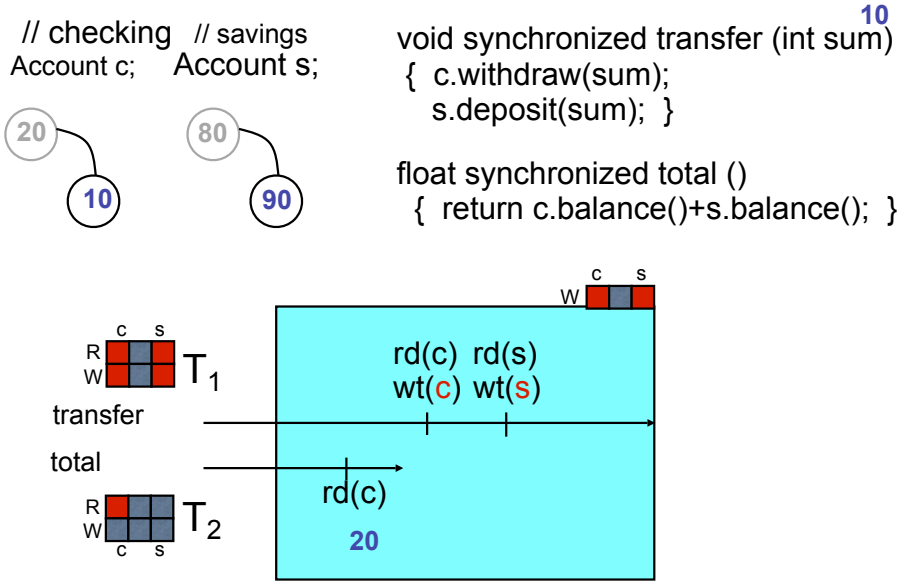
Ensuring Serializability



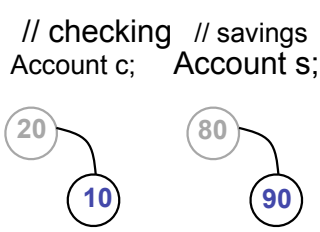
Ensuring Serializability



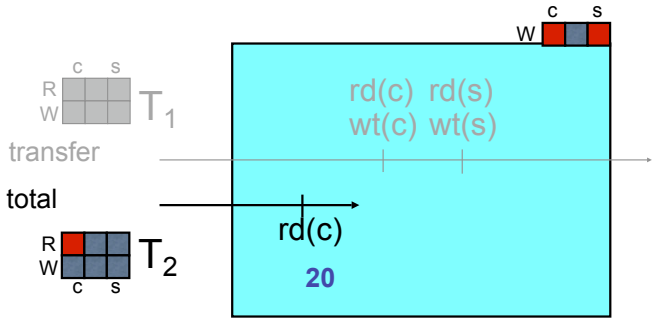
Ensuring Serializability



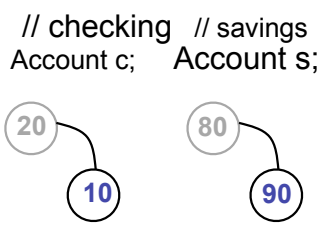
Ensuring Serializability



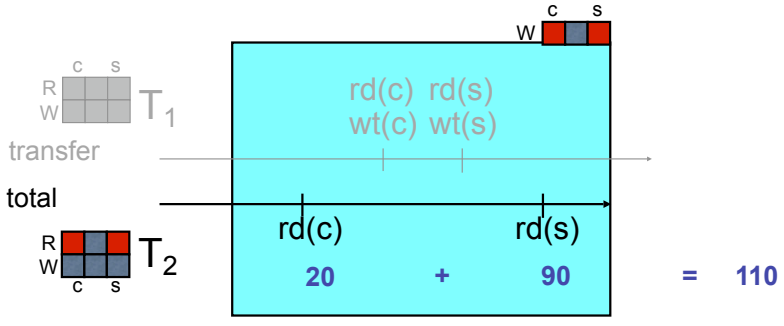

```
10
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



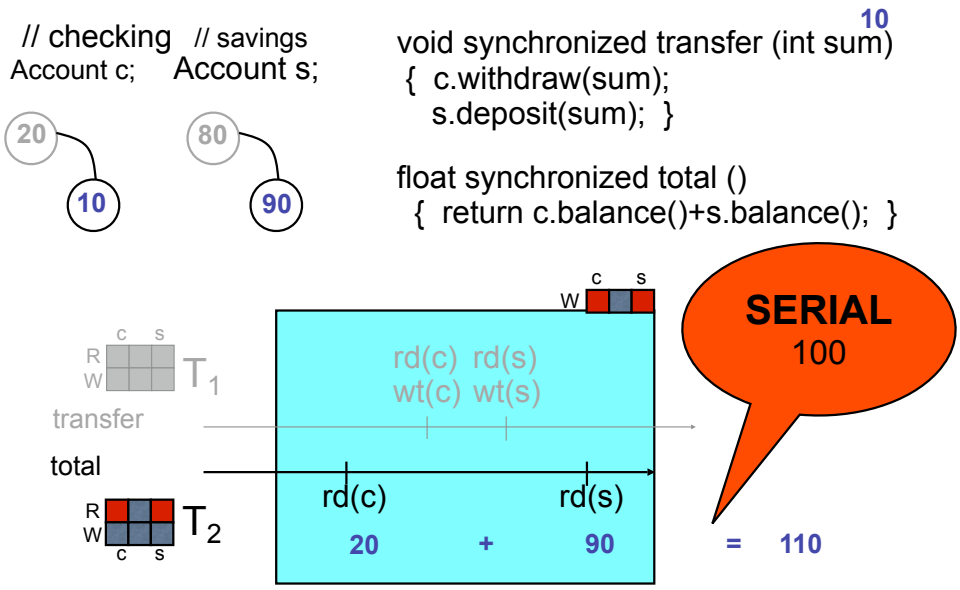
Ensuring Serializability



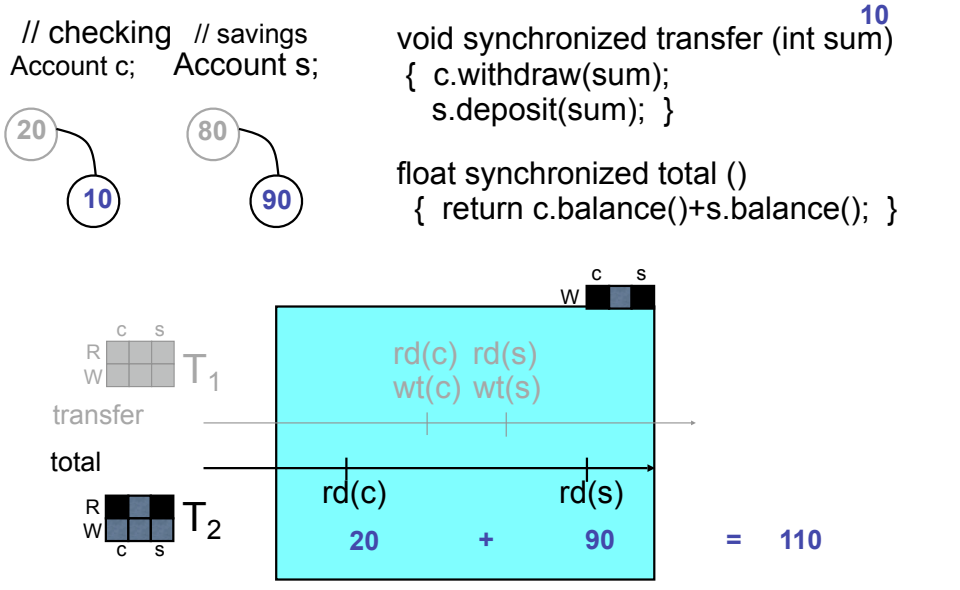

```
10
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



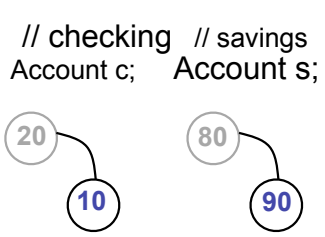
Ensuring Serializability



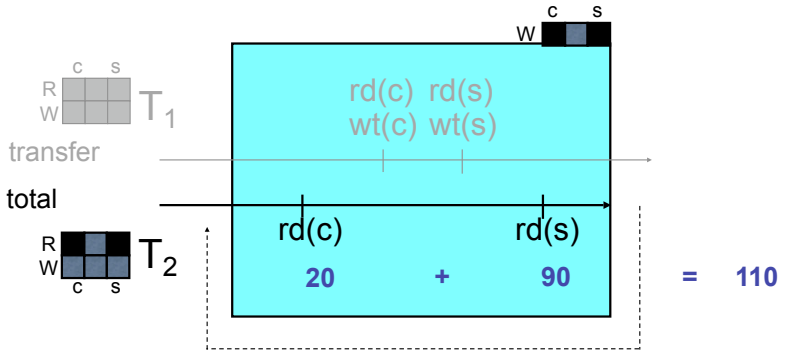
Ensuring Serializability



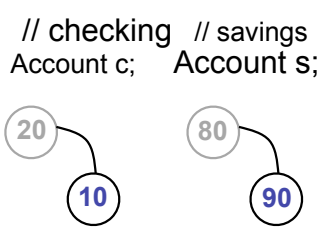
Ensuring Serializability



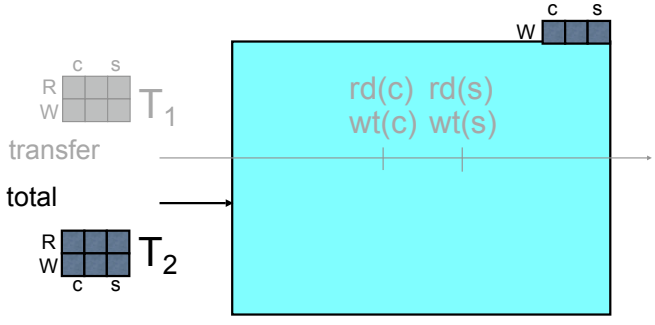

```
10
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



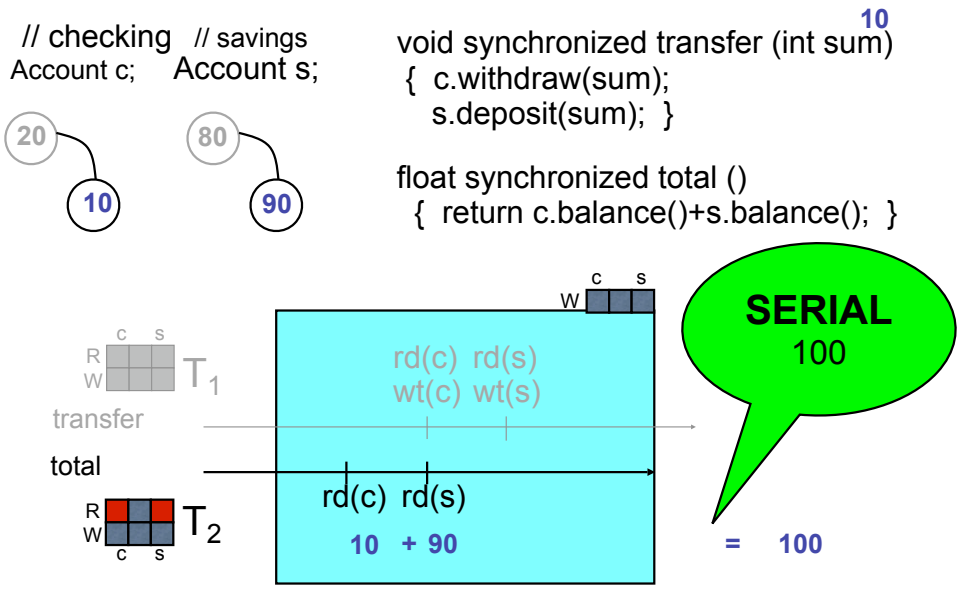
Ensuring Serializability




```
10
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



Ensuring Serializability



Observations

- Classical lock-based approaches to coordinating activities of multiple threads:
  - Impose a heavy burden on programmer to balance safety and performance.
  - Have well-known issues with deadlocks, data races, priority inversion, interaction with external actions, etc.
  - Scalability impacted by the use of mutual-exclusion.
- But ...
  - There is much legacy code (e.g., libraries) that use locks.
  - Well-known tuned implementations.

    Thin locks.

Observations

- Software transactions:
  - Enforce atomicity and isolation on the regions they protect:
    Atomicity: actions within a transaction appear to execute all-at-once or not-at-all.
    Isolation: effects of other threads are not witnessed once a transaction starts.
  - Conceptually simple programming model
- But ...
  - More complicated implementation model.
    Must track atomicity and isolation violations at runtime.
    Revocation of effects when violations occur not always possible.
    Performance benefit only in the presence of contention.