Concurrent Programming

Vitaly Shmatikov

Reading Assignment

- Mitchell, Chapter 14

Concurrency

Two or more sequences of events occur "in parallel"

- **Multiprogramming**
  - Single processor 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
  - Programs on one processor communicate with programs on another
  - Actions may happen simultaneously

Process: sequential program running on a processor

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

Example: Rendering a Web page

- Page is a shared resource
- Multiple concurrent activities in the Web browser
  - Thread for each image load
  - Thread for text rendering
  - Thread for user input (e.g., “Stop” button)
- Cannot all write to page simultaneously!
  - Big challenge in concurrent programming: managing access to shared resources

The Challenges of Concurrency

- Concurrent programs are harder to get right
  - Folklore: need at least an order of magnitude in speedup for concurrent program 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

Language Support for Concurrency

- Threads
  - Think of a thread as a system “object” containing the state of execution of a sequence of function calls
  - Each thread needs a separate run-time stack (why?)
  - Pass threads as arguments, return as function results
- Communication abstractions
  - Synchronous communication
  - Asynchronous buffers that preserve message order
- Concurrency control
  - Locking and mutual exclusion
  - Atomicity is more abstract, less commonly provided

Inter-Process Communication

- Processes may need to communicate
  - Process requires exclusive access to some resources
  - Process need to exchange data with another process
- Can communicate via:
  - Shared variables
  - Message passing
  - Parameters

Explicit vs. Implicit Concurrency

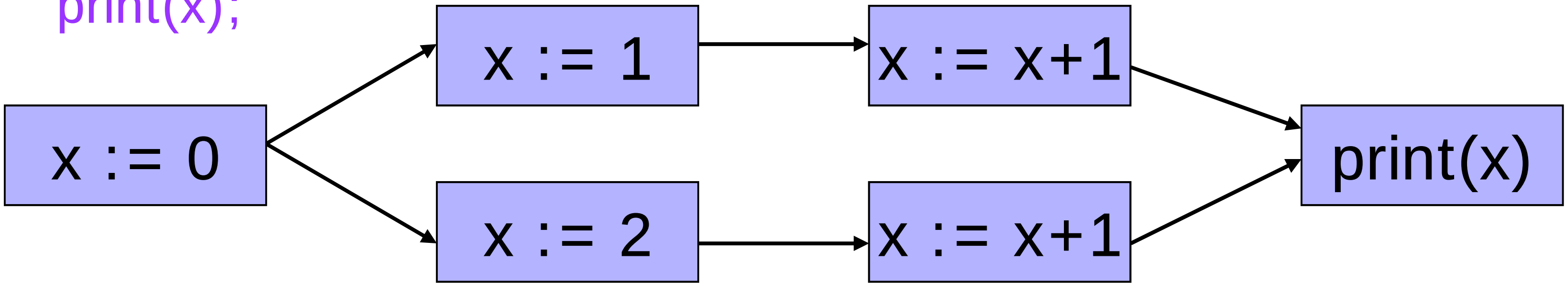
- Explicit concurrency
  - Fork or create threads / processes explicitly
  - Explicit communication between processes
    - Producer computes useful value
    - Consumer requests or waits for producer
- Implicit concurrency
  - Rely on compiler to identify potential parallelism
  - Instruction-level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

cobegin / coend

- Limited concurrency primitive
  – Concurrent Pascal [Per Brinch Hansen, 1970s]

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

execute sequential blocks in parallel



Atomicity at level of assignment statement

Properties of cobegin/coend

- Simple way to create concurrent processes
- Communication by shared variables
- No mutual exclusion
- No atomicity
- Number of processes fixed by program structure
- Cannot abort processes
  - All must complete before parent process can go on

Race Conditions

- Race condition occurs when the value of a variable depends on the execution order of two or more concurrent processes (why is this bad?)
- Example

```
procedure signup(person)
  begin
    number := number + 1;
    list[number] := person;
end;
signup(joe) || signup(bill)
```

Critical Section

- Two concurrent processes may access a shared resource
- Inconsistent behavior if processes are interleaved
- Allow only one process in critical section
- Issues
  - How to select which process is allowed to access the critical section?
  - What happens to the other process?

Locks and Waiting

<initialize concurrency control>

Process 1:

```
    <wait>
    signup(joe);  // critical section
    <signal>
```

Process 2:

```
    <wait>
    signup(bill);   // critical section
    <signal>
```

Need atomic operations to implement wait

Deadlock

- Deadlock occurs when a process is waiting for an event that will never happen
- Necessary conditions for a deadlock to exist:
  - Processes claim exclusive access to resources
  - Processes hold some resources while waiting for others
  - Resources may not be removed from waiting processes
  - There exists a circular chain of processes in which each process holds a resource needed by the next process in the chain
- Example: “dining philosophers”

Implementing Mutual Exclusion

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

Semaphores

- Semaphore is an integer variable and an associated process queue
- Operations:
  - P(s)   if s > 0 then s--
    else enqueue process
  - V(s)   if a process is enqueued then dequeue it
    else s++
- Binary semaphore
- Counting semaphore

Simple Producer-Consumer

```
program SimpleProducerConsumer;
var buffer : string;
    full : semaphore = 0;
    empty : semaphore = 1;
begin
    cobegin
        Producer; Consumer;
    coend;
end.
```

```
procedure Producer;
var tmp : string
begin
   while (true) do begin
      produce(tmp);
      P(empty);  { begin critical section }
      buffer := tmp;
      V(full);   { end critical section }
  end;
end;
```

```
procedure Consumer;
var tmp : string
begin
    while (true) do begin
        P(full);   { begin critical section }
        tmp := buffer;
        V(empty);  { end critical section }
        consume(tmp);
    end;
end;
```

Producer-Consumer

```
program ProducerConsumer;
const size = 5;
var buffer : array[1..size] of string;
    inn      : integer = 0;
    out      : integer = 0;
    lock     : semaphore = 1;
    nonfull  : semaphore = size;
    nonempty : semaphore = 0;  ...
```

```
procedure Producer;
var tmp : string
begin
    while (true) do begin
        produce(tmp);
        P(nonfull);
        P(lock);   { begin critical section }
        inn := inn mod size + 1;
        buffer[inn] := tmp;
        V(lock);   { end critical section }
        V(nonempty);
    end;
end;
```

```
procedure Consumer;
var tmp : string
begin
    while (true) do begin
        P(nonempty);
        P(lock);   { begin critical section }
        out = out mod size + 1;
        tmp := buffer[out];
        V(lock);   { end critical section }
        V(nonfull);
        consume(tmp);
    end;
end;
```

Monitors

- Monitor encapsulates a shared resource (monitor = “synchronized object”)
  - Private data
  - Set of access procedures (methods)
  - Locking is automatic
    - At most one process may execute a monitor procedure at a time (this process is “in” the monitor)
    - If one process is in the monitor, any other process that calls a monitor procedure will be delayed

Example of a Monitor

```
monitor Buffer;
const size = 5;
var buffer : array[1..size] of string;
    in      : integer = 0;
    out     : integer = 0;
    count   : integer = 0;
    nonfull : condition;
    nonempty : condition;   ...
```

```
procedure put(s : string);

begin
    if (count = size) then wait(nonfull);
    in := in mod size + 1;
    buffer[in] := tmp;
    count := count + 1;
    signal(nonempty);
end;
```

```
function get : string;
var tmp : string
begin
    if (count = 0) then wait(nonempty);
    out = out mod size + 1;
    tmp := buffer[out];
    count := count - 1;
    signal(nonfull);
    get := tmp;
end;
```

Java Threads

- Thread
  - Set of instructions to be executed one at a time, in a specified order
  - Special Thread class is part of the core language
    - In C/C++, threads are part of an “add-on” library
- Methods of class Thread
  - start : method called to spawn a new thread
    - Causes JVM to call run() method on object
  - suspend : freeze execution (requires context switch)
  - interrupt : freeze and throw exception to thread
  - stop : forcibly cause thread to halt

java.lang.Thread

```
public class Thread implements Runnable {
      private char name[];
      private Runnable target;
      ...
      public final static int MIN_PRIORITY = 1;
      public final static int NORM_PRIORITY = 5;
      public final static int MAX_PRIORITY = 10;

      private void init(ThreadGroup g, Runnable target, String name) {...}

      public Thread() { init(null, null, "Thread-" + nextThreadNum()); }
      public Thread(Runnable target) {
         init(null, target, "Thread-" + nextThreadNum());
     }
      public Thread(Runnable target, String name) { init(null, target, name); }

      public synchronized native void start();

      public void run() {
         if (target != null
             target.run();
         }
     }
}
```

What does this mean?

Creates execution environment for the thread (sets up a separate run-time stack, etc.)

Methods of Thread Class

```
public class Thread implements Runnable {
      ...
      public static native Thread currentThread();
      public static native void yield();
      public static native void sleep(long millis) throws InterruptedException;
      public static int enumerate(Thread tarray[])

      public static boolean interrupted() { ... }
     public boolean isInterrupted() { ... }
      public final native boolean isAlive();
      public String toString() {
      public void interrupt() { ... }
      public void interrupt() { ... }
      public final void stop() { ... }
      public final void suspend() { ... }
      public final void resume() { ... }
      public final void setPriority(int newPriority) {
      public final int getPriority() {
      public final void setName(String name) { ... }
      public final String getName() { return String.valueOf(name); }
      public native int countStackFrames();
      public final synchronized void join() throws InterruptedException {...}
      public void destroy() { throw new NoSuchMethodError(); }
}
```

Runnable Interface

- Thread class implements Runnable interface
- Single abstract (pure virtual) method run()

```
public interface Runnable {
    public void run(); }
```

- Any implementation of Runnable must provide an implementation of the run() method

```
public class ConcurrentReader implements Runnable {
    ...
    public void run() { ...
        ... code here executes concurrently with caller ... }
}
```

Two Ways to Start a Thread

- Construct a thread with a runnable object

```
ConcurrReader readerThread = new ConcurrReader();
Thread t = new Thread(readerThread);
t.start();   // calls ConcurrReader.run() automatically
```

... OR ...

- Instantiate a subclass of Thread

```
class ConcurrWriter extends Thread { ...
    public void run() { ... } }
ConcurrWriter writerThread = new ConcurrWriter();
writerThread.start();   // calls ConcurrWriter.run()
```

Why Two Ways?

- Java only has single inheritance
- Can inherit from some class, but also implement Runnable interface so that can run as a thread

```
class X extends Y implements Runnable { ...
    public synchronized void doSomething() { ... }
    public void run() { doSomething(); }
}
X obj = new X();
obj.doSomething(); // runs sequentially in current thread
Thread t = new Thread(new X()); // new thread
t.start();   // calls run() which calls doSomething()
```

Interesting “Feature”

[Allen Holub, “Taming Java Threads”]

- Java language specification allows access to objects that have not been fully constructed

```
class Broken {
    private long x;
    Broken() {
        new Thread() {
            public void run() { x = -1; }
        }.start();
        x = 0;
} }
```

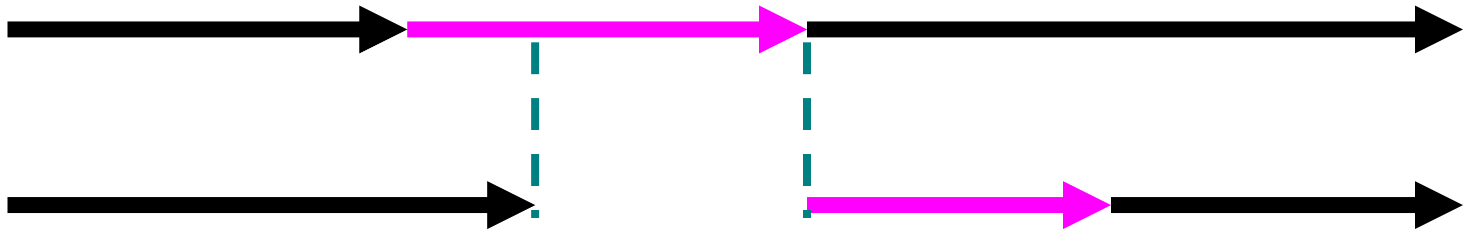
Thread created within constructor can access partial object

Interaction Between Threads

- Shared variables and method calls
  - Two threads may assign/read the same variable
    - Programmer is responsible for avoiding race conditions by explicit synchronization!
  - Two threads may call methods on the same object
- Synchronization primitives
  - All objects have an internal lock (inherited from Object)
  - Synchronized method locks the object
    - While it is active, no other thread can execute inside object
  - Synchronization operations (inherited from Object)
    - Wait: pause current thread until another thread calls Notify
    - Notify: wake up waiting thread

Synchronized Methods

- Provide mutual exclusion
  - If a thread calls a synchronized method, object is locked
  - If another thread calls a synchronized method on the same object, this thread blocks until object is unlocked
    - – Unsynchronized methods can still be called!



- "synchronized" is not part of method signature
  - Subclass may replace a synchronized method with unsynchronized method

Wait, Notify, NotifyAll

```
public class Object {
      ...
      public final native void notify();
      public final native void notifyAll();

      public final native void wait(long timeout) throws InterruptedException;
      public final void wait() throws InterruptedException { wait(0); }
      public final void wait(long timeout, int nanos)
         throws InterruptedException { ... }
}
```

- wait() releases object lock, thread waits on internal queue
- notify() wakes the highest-priority thread closest to the front of the object's internal queue
- notifyAll() wakes up all waiting threads
  - Threads non-deterministically compete for access to object
  - May not be fair (low-priority threads may never get access)
- May only be called when object is locked (when is that?)

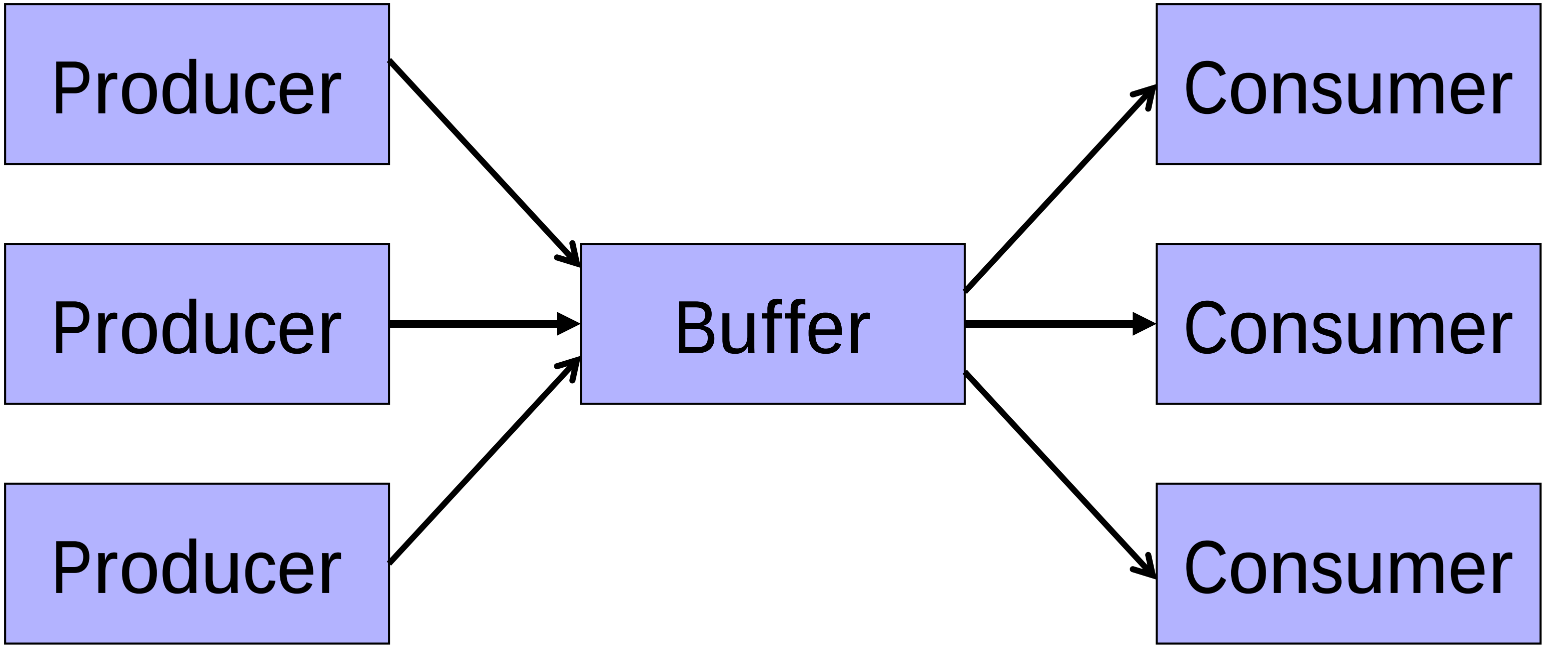
Using Synchronization

```
public synchronized void consume() {
    while (!consumable()) {
        wait(); }    // release lock and wait for resource
    ...   // have exclusive access to resource, can consume
}

public synchronized void produce() {
    ... // do something that makes consumable() true
    notifyAll();    // tell all waiting threads to try consuming
    //  can also call notify() and notify one thread at a time
}
```

Example: Shared Queue

```
class SharedQueue {
    private Element head, tail;

    public boolean empty() { return head == tail; }

    public synchronized Element remove() {
      try { while (empty()) wait(); } // wait for an element in the queue
      catch (InterruptedException e) { return null; }
      Element p = head; head = head.next;
      if (head == null) tail == null;
      return p;
    }

    public synchronized void insert(Element p)
      if (tail == null) head = p;
      else tail.next = p;
      p.next = null;
      tail = p;
      notify();  // let one waiter know something is in the queue
    }
}
```

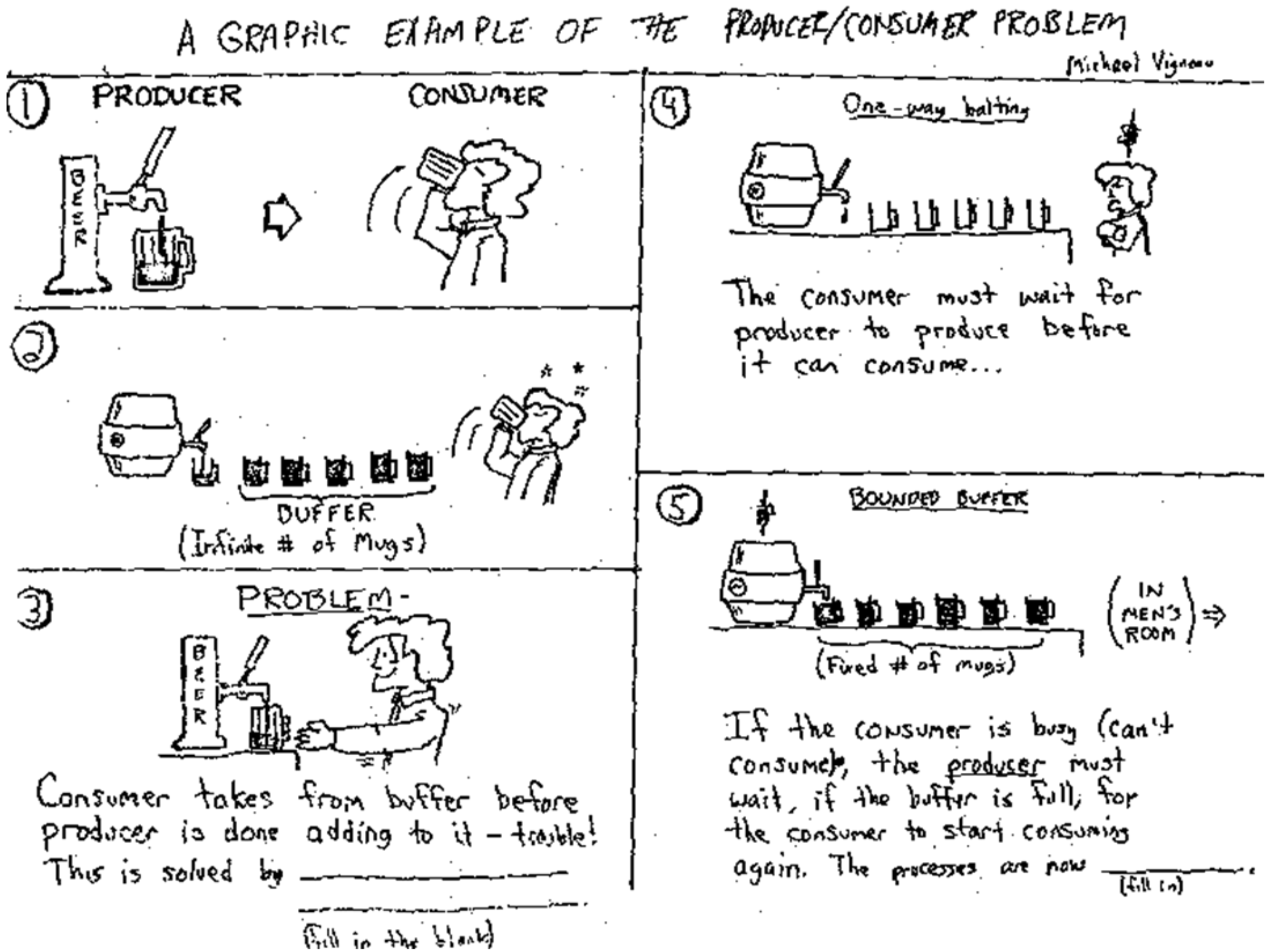
Example: Producer-Consumer



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

In Pictures

[from Jeffrey Smith]



Solving Producer-Consumer

- Cannot be solved with locks alone
- Consumer must wait until buffer is not empty
  - While waiting, must sleep (use wait method)
  - Need condition recheck loop
- Producer must inform waiting consumers when there is something in the buffer
  - Must wake up at least one consumer (use notify method)

Implementation in Stack<T>

```
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; }
```

Why is loop needed here?

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(); }
```

- Need a loop because another process may run instead

```
public synchronized void lock() throws InterruptedException {
         while ( isLocked ) wait();
        isLocked = true; }
```

Nested Monitor Lockout Problem

- Wait and notify used within synchronized code
  - Purpose: make sure that no other thread has called method of same object
- Wait causes the thread to give up its lock and sleep until notified
  - Allow another thread to obtain lock and continue processing
- Calling a blocking method within a synchronized method can lead to deadlock

Nested Monitor Lockout Example

```
class Stack {
    LinkedList list = new LinkedList();
    public synchronized void push(Object x) {
    synchronized(list) {
            list.addLast( x ); notify();
    } }
    public synchronized Object pop() {
    synchronized(list) {
            if( list.size() <= 0 ) wait();
            return list.removeLast();
    } }
}
```

Could be blocking method of List class

Releases lock on Stack object but not lock on list; a push from another thread will deadlock

Preventing Nested Monitor Deadlock

- No blocking calls in synchronized methods, OR
- Provide some nonsynchronized method of the blocking object

- No simple solution that works for all programming situations

Synchronized Blocks

- Any Java block can be synchronized

```
synchronized(obj) {
   ... mutual exclusion on obj holds inside this block ...
}
```

- Synchronized method declaration is just syntactic sugar for syncronizing the method's scope

```
public synchronized void consume() { ... body ... }
```

is the same as

```
public void consume() {
      synchronized(this) { ... body ... }
  }
```

Locks Are Recursive

- A thread can request to lock an object it has already locked without causing deadlock

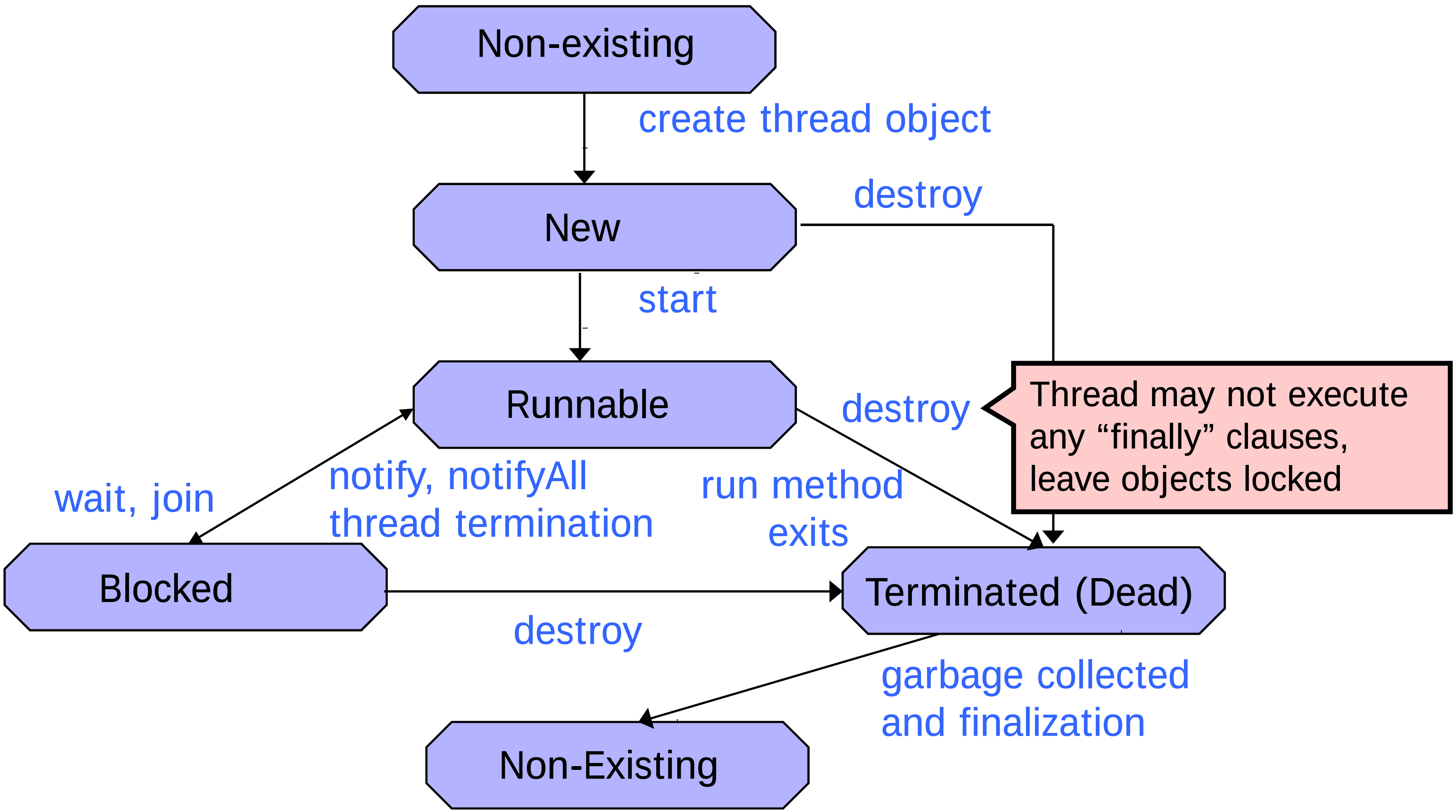
```
public class Foo {
    public void synchronized f() { ... }
    public void synchronized g() { ... f(); ... }
}

Foo f = new Foo;
synchronized(f) { ... synchronized(f) { ... } ... }
```

Synchronizing with Join()

- Join() waits for thread to terminate

```
class Future extends Thread {
    private int result;
    public void run() {  result = f(...); }
    public int getResult() { return result;}
}
...
Future t = new future;
t.start()                              // start new thread
...
t.join(); x = t.getResult();   // wait and get result
```

States of a Java Thread



Concurrent Garbage Collection

- Need to stop thread while mark-and-sweeping
  - Do other threads need to be stopped?
- Problem: objects may change during collection
- Solution: prevent read/write to memory area
  - Subtle!
  - Generational GC distinguishes short-lived and long-lived objects
  - Copying collectors allows reads from old area if writes are blocked...

Limitations of Java 1.4 Primitives

- Cannot back off an attempt to acquire a lock
  - Can’t give up after waiting for a certain period of time or after an interrupt
- Cannot alter the semantics of a lock
  - Reentrancy, read versus write protection, fairness, ...
- No access control for synchronization
  - Any method can do synchronized(obj) on any object
- Synchronization limited to block-structured locking
  - Can’t acquire a lock in one method, release in another

POSIX Threads

- Pthreads library for C

pthread_create - create a new thread giving it a “starting” procedure to run along with a single argument.
pthread_self - ask the currently running thread for its thread id.
pthread_join - join with a thread using its thread id (an integer value)

pthread_mutex_init - initialize a mutex structure
pthread_mutex_destroy - destroy a mutex structure
pthread_mutex_lock - lock an initialized mutex, if already locked suspend execution and wait
pthread_mutex_trylock - try to lock a mutex and if unsucessful, do not suspend execution
pthread_mutex_unlock - unlock a mutex that was locked by the current thread

pthread_cond_init - initialize a condition variable structure
pthread_cond_destroy - destroy a condition variable structure
pthread_cond_wait - block the currently running thread on a condition variable indefintely
pthread_cond_timedwait - block the currently running thread on a condition variable for a specific time
pthread_cond_signal - wakeup one thread blocked on a condition variable
pthread_cond_broadcast - wakeup all threads blocked on a condition variable

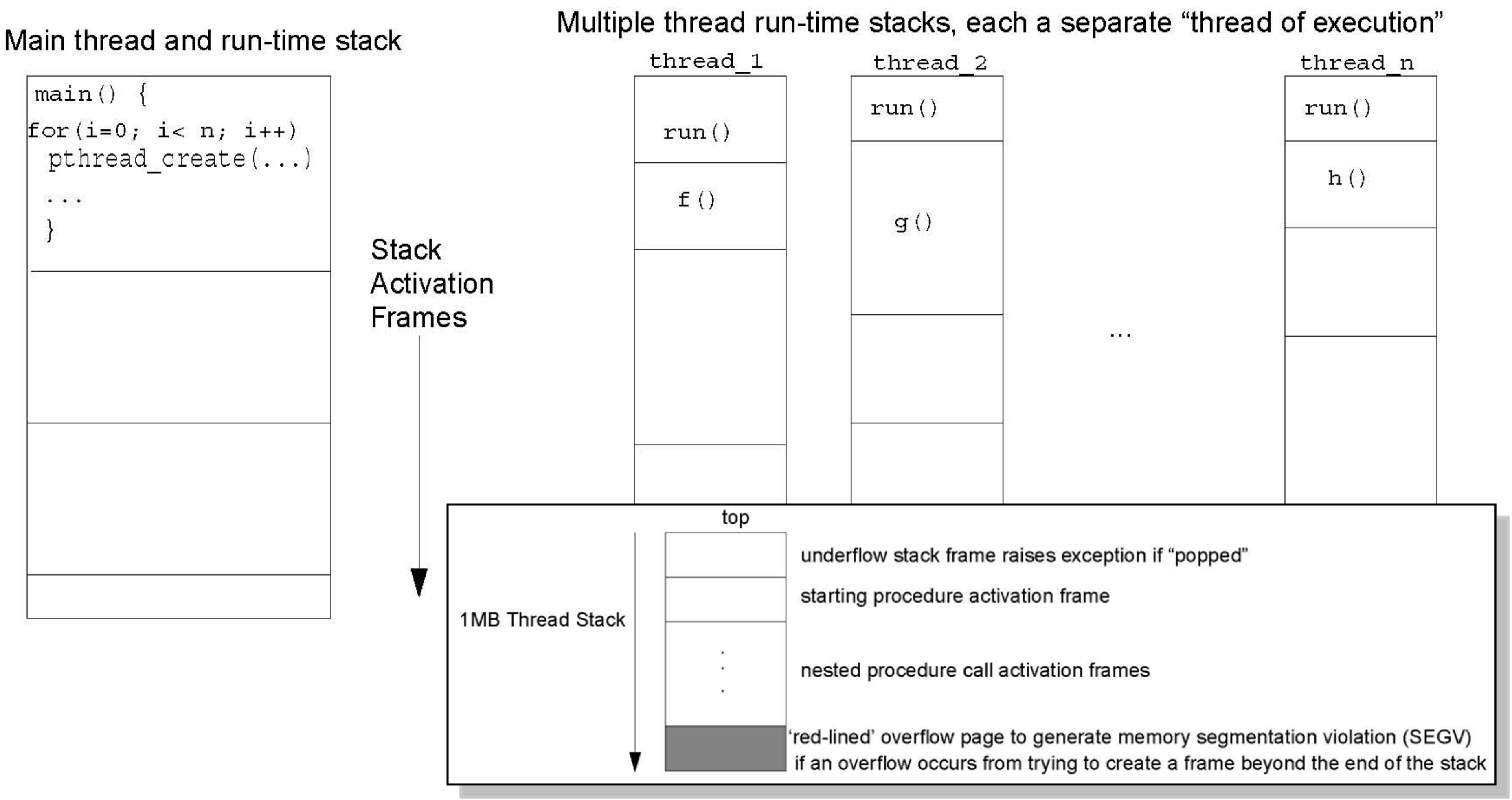
Example of Using POSIX Threads

```
#include <pthread.h>
#include <unistd.h>  /* sleep declaration */
#include <stdio.h>  /* printf declaration */
const int NUM_THREADS = 5;

void* sleeping(void* st)
{
    int sleep_time = (int) st; /* cast void* to an int */
    printf (“thread %d sleeping %d seconds ...\n”, pthread_self(), sleep_time);
    sleep(sleep_time);
    printf (“\nthread %d awakening\n”, pthread_self());
}

main( int argc, char *argv[] )
{
     pthread_t tid[NUM_THREADS];      /* array of thread IDs */
     int i;

     for ( i = 0; i < NUM_THREADS; i++)
       pthread_create (&tid[i], NULL, sleeping, i+2);

     for ( i = 0; i < NUM_THREADS; i++)
       pthread_join (tid[i], NULL);

     printf (“main() reporting that all %d threads have terminated\n”, i);
}  /* main */
```

Create several child threads

Wait for children to finish

Thread Stacks



Java-Style Synchronization in C++

```
class Synchronized {
    pthread_mutex_t m;  // mutex variable
    pthread_cond_t  c;  // condition variable
protected:

    /* use this class to associate the mutex lock/unlock with the scope of a procedure */
    class Scope {
        Synchronized* obj;
    public:
        Scope(Synchronized* s) : obj(s)  { pthread_mutex_lock(&obj->m); }
        ~Scope() { pthread_mutex_unlock(&obj->m); }
    };

public:

    Synchronized() { // initialize the mutex and condvar on construction
       pthread_mutex_init(&m, 0);
       pthread_cond_init(&c, 0);
    }

    ~Synchronized() { // destroy the mutex and condvar on destruction
        pthread_mutex_destroy(&m);
        pthread_cond_destroy(&c);
    }

    // map Java-like wait, notify and notifyAll onto pthread equivalents

    void wait() { pthread_cond_wait(&c, &m); }
    void notify() { pthread_cond_signal(&c); }
    void notifyAll() { pthread_cond_broadcast(&c); }
};
```

Using C++ Threads

```
class MySynchornizedClass : public Synchronized {
   .. // private instance variables
public:

      // when this classes constructor is called, it first invokes the
      // constructor of the Synchronized class, which initialized the
      // the mutex and condition variable by calling the corresponding
      // pthread_{mutex,cond}_init library procedures
      MySynchronizedClass() { ... }

      // Likewise on destruction, the destructor of the Synchronized class is
      // automatically called and it destroys the mutex and condition variable
      ~MySychronizedClass() { ...}

      // to make a method "synchronized" we declare a local variable of type
      // Synchronized::Scope, which locks the mutex on entry to the procedure scope
      // and automatically unlocks the mutex on exit from the procedure scope

      int some_method(...)
      {
         Synchronized::Scope mx(this);  // automatically locks the mutex
        ... // execute code under mutual exclusion
      } // mx is automatically destructed, which unlocks the mutex
      ...
};
```

Thread Safety of Classes

- Fields of an object or class must always be in a valid state, even when used concurrently by multiple threads
  - What's a "valid state"? Serializability ...
- Classes are designed so that each method preserves state invariants on entry and exit
  - Example: priority queues represented as sorted lists
  - If invariant fails in the middle of a method call, concurrent execution of another method call will observe an inconsistent state

Example: RGBColor Class

```
public class RGBColor {
    private int r; private int g; private int b;
    public RGBColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r; this.g = g; this.b = b;
    }

private static void checkRGBVals(int r, int g, int b) {
        if (r < 0 || r > 255 || g < 0 || g > 255 ||
            b < 0 || b > 255) {
            throw new IllegalArgumentException();
        }
    }
}
```

What goes wrong with multi-threaded use of this class?

```
public void setColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r; this.g = g; this.b = b;
    }

public int[] getColor() {
        //  returns array of three ints: R, G, B
        int[] retVal = new int[3];
        retVal[0] = r;
        retVal[1] = g;
        retVal[2] = b;
        return retVal;
    }

public void invert() {
        r = 255 - r; g = 255 - g; b = 255 - b;
    }
```

Problems with RGBColor Class

- Write/write conflicts
  - If two threads try to write different colors, result may be a “mix” of R,G,B from two different colors
- Read/write conflicts
  - If one thread reads while another writes, the color that is read may not match the color before or after

Making Classes Thread-Safe

- Synchronize critical sections
  - Make fields private, synchronize access to them
- Make objects immutable
  - State cannot be changed after object is created

```
public RGBColor invert() {
   RGBColor retVal = new RGBColor(255 - r, 255 - g, 255 - b);
   return retVal; }
```

  - Examples: Java String and primitive type wrappers Integer, Long, Float, etc.
  - Pure functions are always re-entrant!
- Use a thread-safe wrapper

Thread-Safe Wrapper

- Define new class which has objects of original class as fields, provides methods to access them

```
public synchronized void setColor(int r, int g, int b) {
    color.setColor(r, g, b);
}
public synchronized int[] getColor() {
    return color.getColor();
}
public synchronized void invert() {
    color.invert();
}
```

Comparison

- Synchronizing critical sections
  - Good way to build thread-safe classes from scratch
  - Only way to allow wait() and notify()
- Using immutable objects
  - Good if objects are small, simple abstract data types
  - Benefits: pass without aliasing, unexpected side effects
- Using wrapper objects
  - Works with existing classes, gives users choice between thread-safe version and original (unsafe) one
    - Example: Java 1.2 collections library – classes not thread-safe, but some have methods to enclose objects in safe wrapper

Why Not Synchronize Everything?

- Performance costs
  - Current Sun JVM – synchronized methods are 4 to 6 times slower than non-synchronized
- Risk of deadlock from too much locking
- Unnecessary blocking and unblocking of threads can reduce concurrency
- Alternative: immutable objects
  - Issue: often short-lived, increase garbage collection

Inheritance Anomaly

- Inheritance and concurrency do not mix well
  - Inheritance anomaly identified in 1993 (before Java)
  - Arises in different languages, to different degrees, depending on concurrency primitives
- Concurrency control in derived classes requires redefinition of base class and parents
  - Concurrency control = synchronization, waiting, etc.
- Modification of class requires modifications of seemingly unrelated features in parent classes

Examples of Inheritance Anomaly

- Partitioning of acceptable states
  - Method can only be entered in certain states (enforced by base class)
  - New method in derived class changes set of states
  - Must redefine base class method to check new states
- History-sensitive method entry
  - New method in derived class can only be called after other calls
  - Must modify existing methods to keep track of history

Example: Buffer Class

```
public class Buffer {
   protected Object[] buf;      protected int MAX;    protected int current = 0;
   Buffer(int max) {
       MAX = max;
       buf = new Object[MAX]; }
   public synchronized Object get()  throws Exception {
      while (current<=0) { wait(); }
      current--;
      Object ret = buf[current];
      notifyAll();
      return ret; }
   public synchronized void put(Object v) throws Exception {
      while (current>=MAX) { wait(); }
      buf[current] = v;
      current++;
      notifyAll(); } }
```

Problems in Derived Class

```
public class HistoryBuffer extends Buffer {
  boolean afterGet = false;
  public HistoryBuffer(int max) { super(max); }

  public synchronized Object gget()  throws Exception {
    while ((current<=0)||(!afterGet)) { wait(); }
    afterGet = false;
    return super.get(); }
  public synchronized Object get()  throws Exception {
    Object o = super.get();
    afterGet = true;
    return o; }
  public synchronized void put(Object v) throws Exception {
    super.put(v);
    afterGet = false; } }
```

New method, can be called only after get

Must be redefined to keep track of last method called

Need to redefine to keep track of last method called

util.concurrent

- Doug Lea's utility classes
  - A few general-purpose interfaces
  - Implementations tested over several years
- Principal interfaces and implementations
  - Sync: acquire/release protocols
  - Channel: put/take protocols
  - Executor: executing Runnable tasks

Sync

- Main interface for acquire/release protocols
  - Used for custom locks, resource management, other common synchronization idioms
  - Coarse-grained interface, doesn't distinguish different lock semantics
- Implementations
  - Mutex, ReentrantLock, Latch, CountDown, Semaphore, WaiterPreferenceSemaphore, FIFOSemaphore, PrioritySemaphore
  - ObservableSync, LayeredSync to simplify composition and instrumentation

Channel

- Main interface for buffers, queues, etc.

- Implementations
  - LinkedQueue, BoundedLinkedQueue, BoundedBuffer, BoundedPriorityQueue, SynchronousChannel, Slot

Executor

- Main interface for Thread-like classes
  - Pools
  - Lightweight execution frameworks
  - Custom scheduling
- Need only support execute(Runnable r)
  - Analogous to Thread.start
- Implementations
  - PooledExecutor, ThreadedExecutor, QueuedExecutor, FJTaskRunnerGroup
  - Related ThreadFactory class allows most Executors to use threads with custom attributes

java.util.Collection

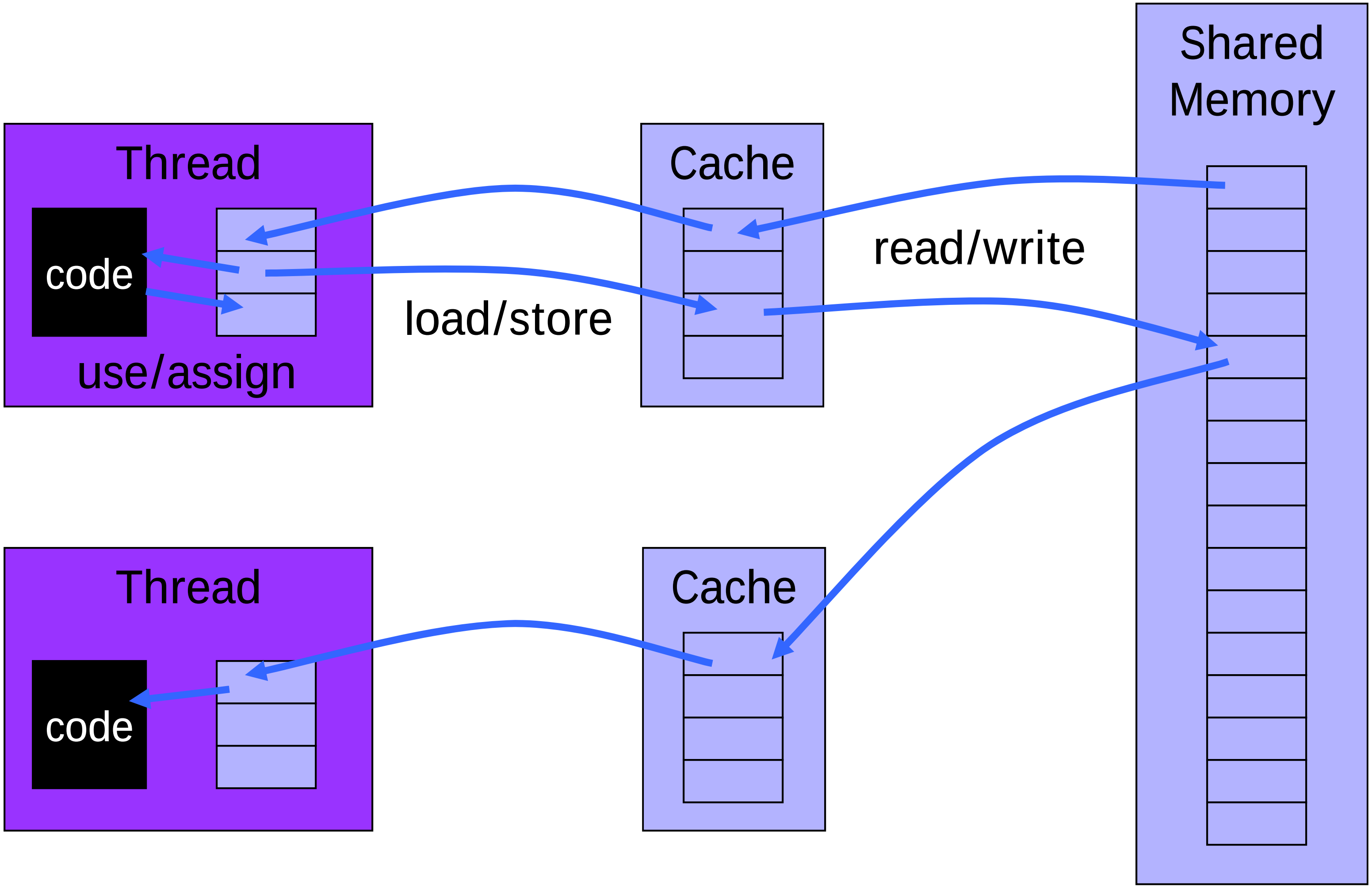
- Adapter-based scheme
  - Allow layered synchronization of collection classes
- Basic collection classes are unsynchronized
  - Example: java.util.ArrayList
  - Except for Vector and Hashtable
- Anonymous synchronized Adapter classes
  - Constructed around the basic classes, e.g.,

    `List l = Collections.synchronizedList(new ArrayList());`

Java Memory Model

- Multithreaded access to shared memory
  - Competitive threads access shared data
  - Can lead to data corruption
- Memory model determines:
  - Which program transformations are allowed
    - Should not be too restrictive
  - Which program outputs may occur on correct implementation
    - Should not be too generous
  - Need semantics for incorrectly synchronized programs

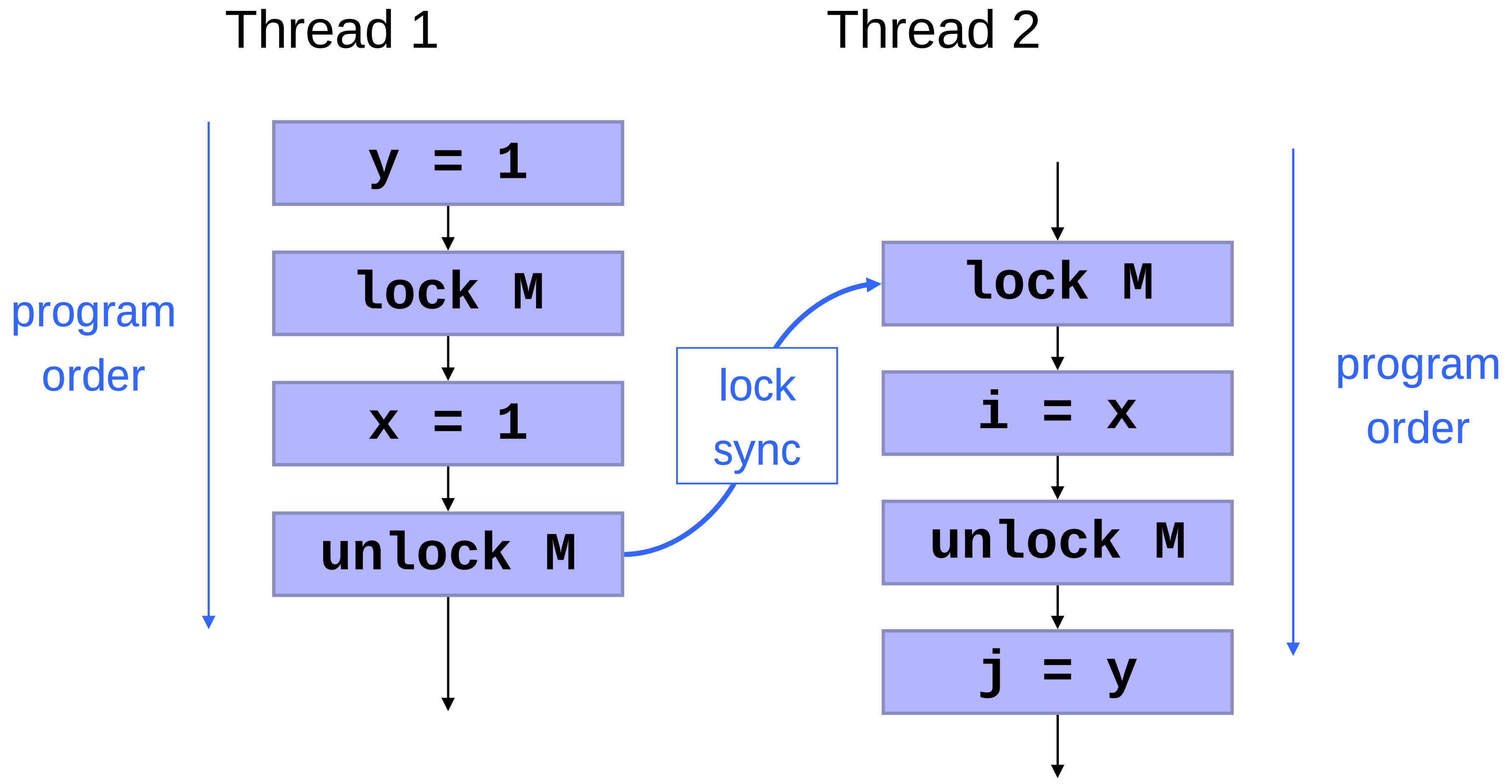
Memory Hierarchy



Old memory model placed complex constraints on read, load, store, etc.

Program and Locking Order

[Manson, Pugh]



Race Conditions

- “Happens-before” order
  - Transitive closure of program order and synchronizes-with order (what does this mean?)
    - Program order as written or as compiled and optimized?
- Conflict
  - An access is a read or a write
  - Two accesses conflict if at least one is a write
- Race condition
  - Two accesses form a data race if they are from different threads, they conflict, and they are not ordered by happens-before

Races in Action

- Northeast Blackout of 2003
  - Affected 50 million people in U.S. and Canada
- Race condition in alarm management system caused it to stall, alarms backed up and stalled both primary and backup server
  - “We had in excess of three million online operational hours in which nothing had ever exercised that bug. I'm not sure that more testing would have revealed it.”

    -- GE Energy's Mike Unum

Memory Model Question

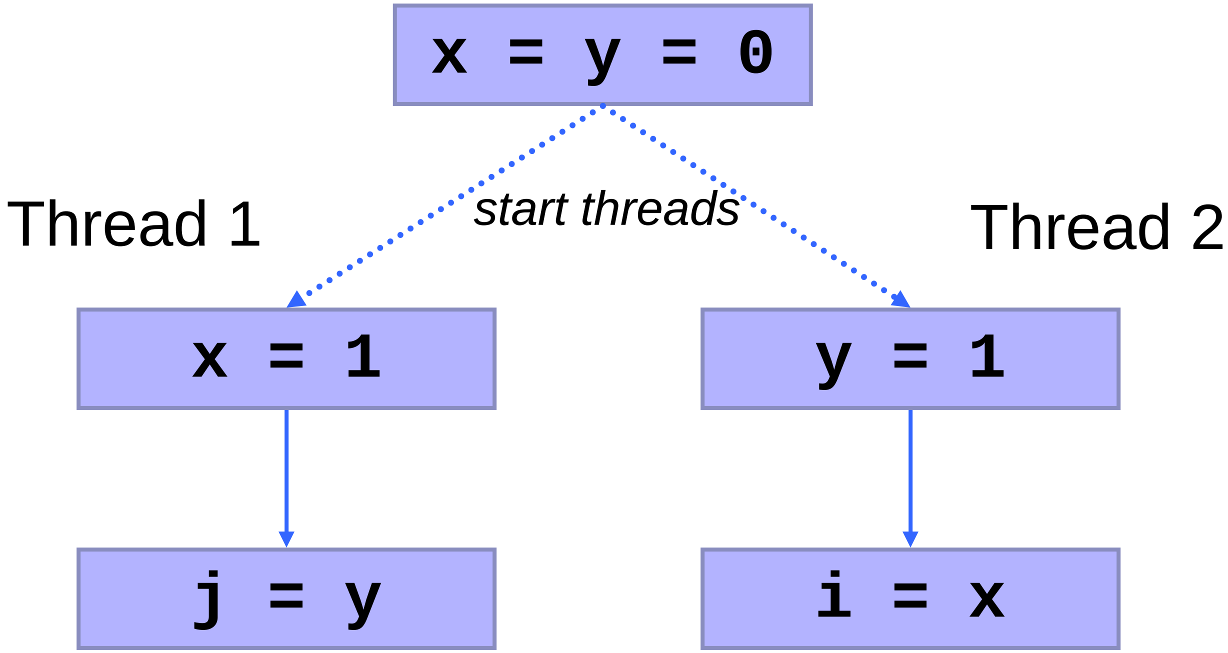
- How should the compiler and run-time system be allowed to schedule instructions?
- Possible partial answer
  - If instruction A occurs in Thread 1 before release of lock, and B occurs in Thread 2 after acquire of same lock, then A must be scheduled before B
- Does this solve the problem?
  - Too restrictive: if no reordering allowed in threads
  - Too permissive: if arbitrary reordering in threads
  - Compromise: allow local thread reordering that would be OK for sequential programs

Instruction Reordering

- Compilers can reorder instructions
  - If two instructions are independent, do in any order
  - Take advantage of registers, etc.
- Correctness for sequential programs
  - Observable behavior should be same as if program instructions were executed in the order written
- Sequential consistency for concurrent programs
  - If program has no data races, then memory model should guarantee sequential consistency
  - What about programs with races?
    - Reasonable programs may have races (need to test, debug, ...)

Example Program with Data Race

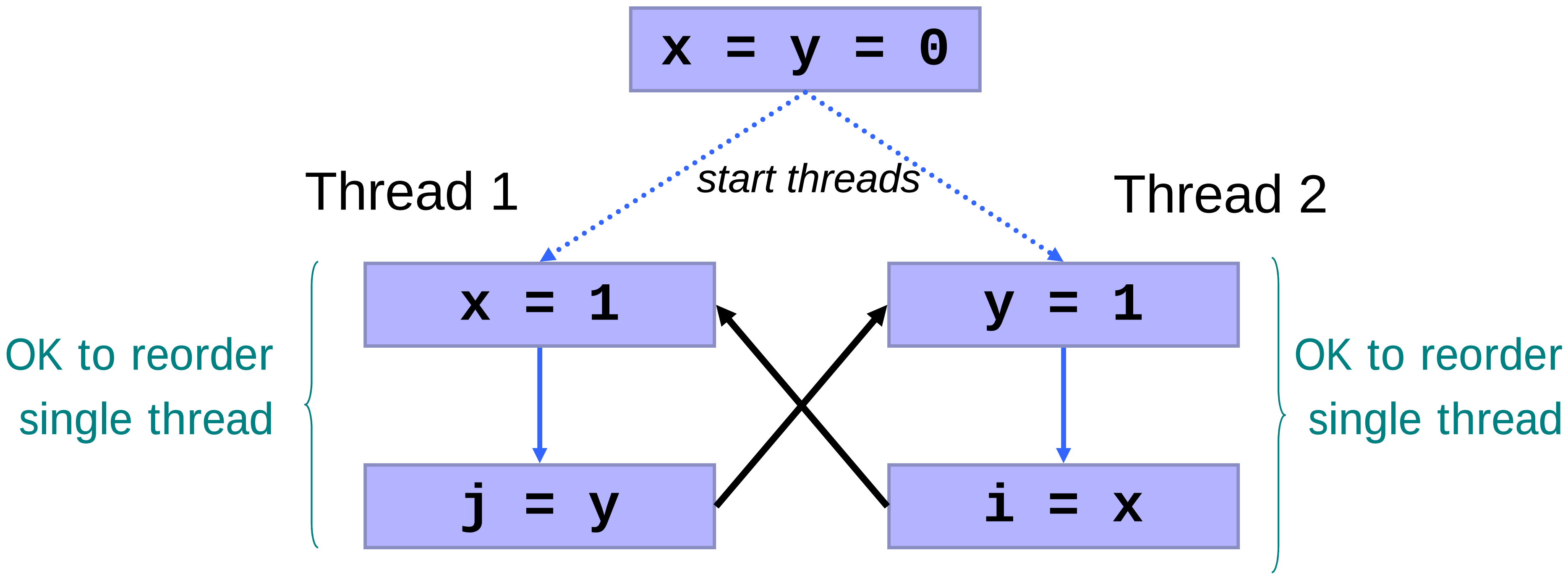
[Manson, Pugh]



Can we end up with i = 0 and j = 0?

Sequential Reordering + Data Race

[Manson, Pugh]



Can we end up with i = 0 and j = 0? Yes!

Java definition considers this OK since there is a data race

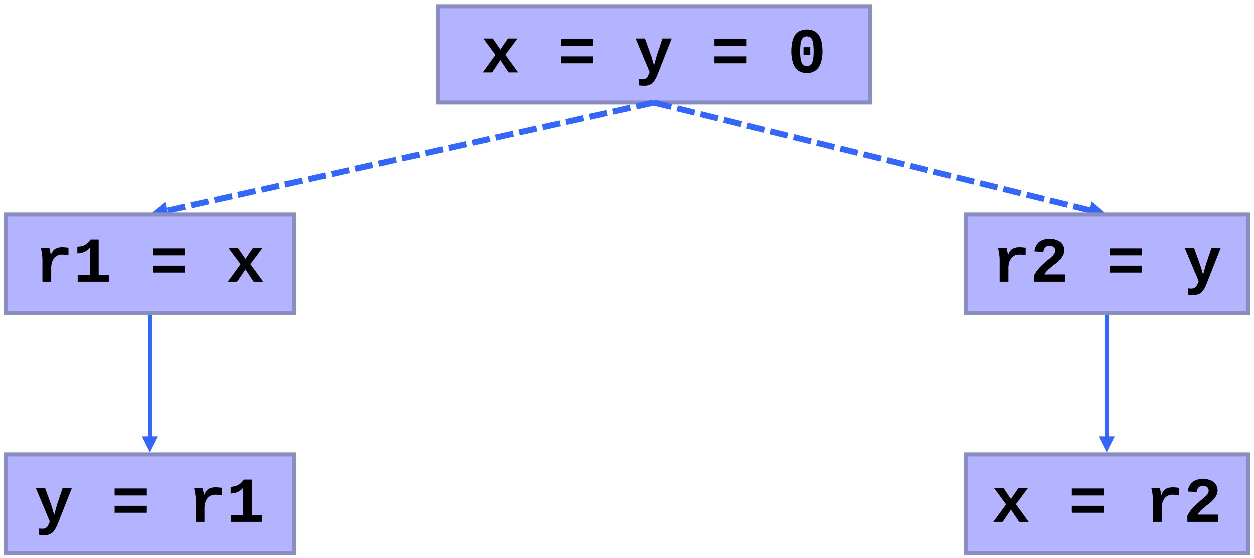
Allowed Sequential Reordering

[Manson, Pugh]

- "Roach motel" ordering
  - Compiler/processor can move accesses into synchronized blocks
  - Can only move them out under special circumstances, generally not observable
- Release only matters to a matching acquire
- Special cases:
  - Locks on thread local objects are a no-op
  - Reentrant locks are a no-op
- Java SE 6 (Mustang) optimizes based on this

Want To Prevent This

[Manson, Pugh]



- Must not result in r1 = r2 = 42
  - Imagine if 42 were a reference to an object!
- Value appears “out of thin air”
  - Causality run amok
  - Legal under a simple “happens-before” model of possible behaviors

Summary of Memory Model

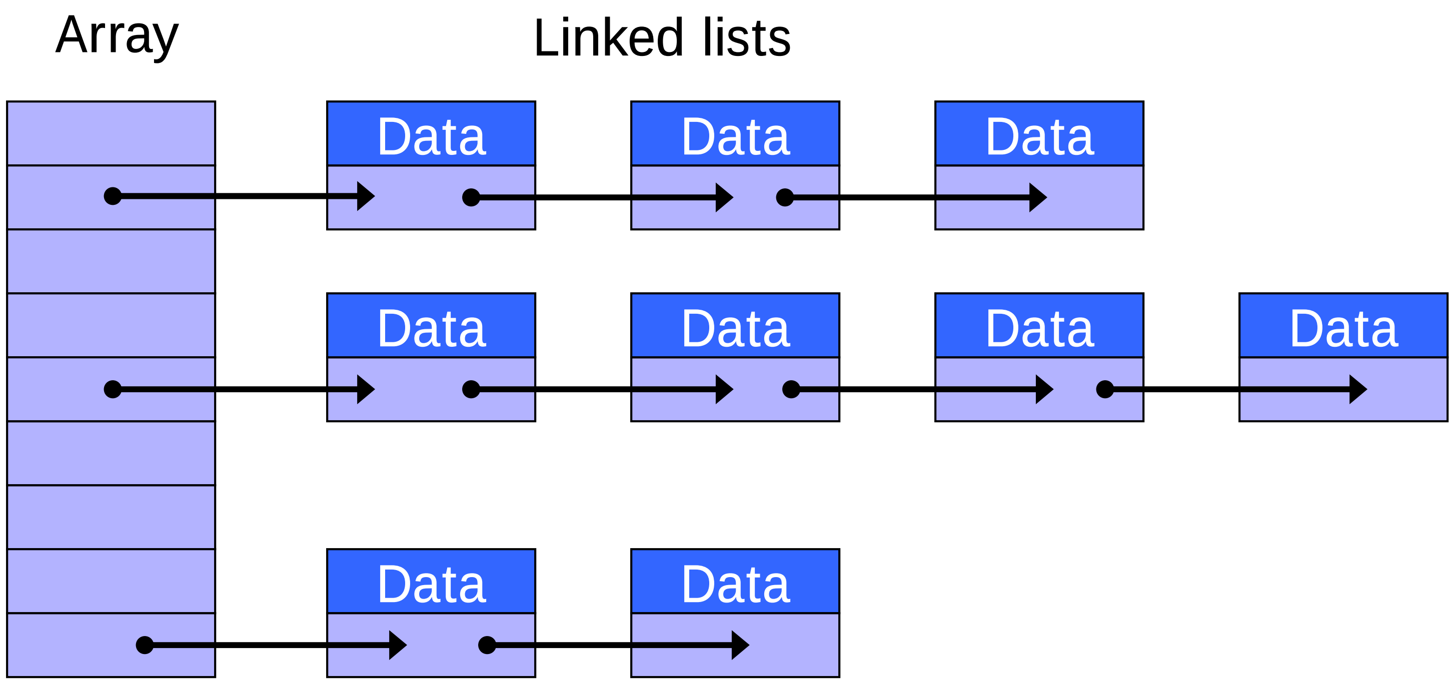
- Strong guarantees for race-free programs
  - Equivalent to interleaved execution that respects synchronization actions
  - Reordering must preserve thread's sequential semantics
- Weaker guarantees for programs with races
  - No weird out-of-the-blue program results
  - Allows program transformation and optimization
- Form of actual memory model definition
  - Happens-before memory model
  - Additional condition: for every action that occurs, there must be identifiable cause in the program

Example: Concurrent Hash Map

- Implements a hash table
  - Insert and retrieve data elements by key
  - Two items in same bucket placed in linked list
- Tricky

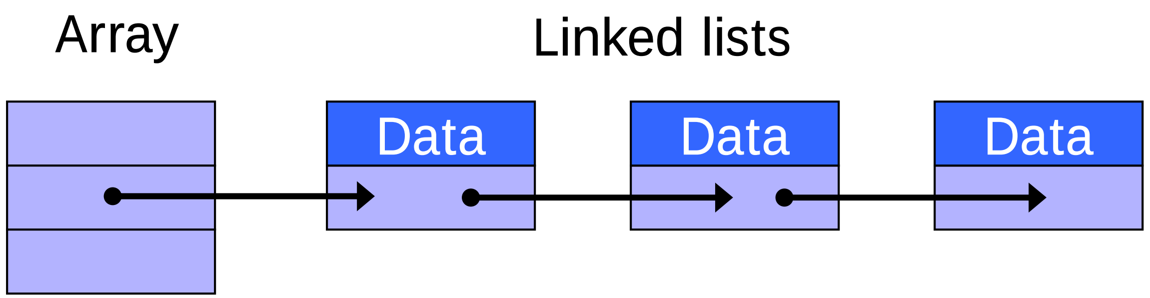
  "ConcurrentHashMap is both a very useful class for many concurrent applications and a fine example of a class that understands and exploits the subtle details of the Java Memory Model (JMM) to achieve higher performance. ... Use it, learn from it, enjoy it – but unless you're an expert on Java concurrency, you probably shouldn't try this on your own."

ConcurrentHashMap



- Concurrent operations
  - read: no problem
  - read/write: OK if different lists
  - read/write to same list: clever tricks sometimes avoid locking

ConcurrentHashMap Tricks



- List cells immutable, except for data field
  - Read thread sees a linked list, even if concurrent write in progress
- Add to list by inserting at the head
- Remove from list: set data field to null, rebuild list to skip this cell
  - Unreachable cells eventually garbage collected

Atomicity

- Mark block so that compiler and run-time system will execute it without interaction from other threads
- Advantages
  - Simple, powerful correctness property
  - Stronger than race freedom (why?)
  - Enables sequential reasoning

Limitations of Race-Freedom (1)

[Flanaghan]

```
class Ref {
  int i;
  void inc() {
    int t;
    synchronized (this) {
      t = i;
    }
    synchronized (this) {
      i = t+1;
    }
  }
  ...
}
```

Ref.inc()

- Race-free
- Behaves incorrectly in a multithreaded context

Race freedom does not prevent errors due to unexpected interactions between threads

Limitations of Race-Freedom (2)

[Flanaghan]

```
class Ref {
  int i;
  void inc() {
    int t;
    synchronized (this) {
      t = i;
      i = t+1;
    }
  }

  void read() { return i; }
  ...
}
```

Ref.read()

- Has a race condition
- Behaves correctly in a multithreaded context

Race freedom <u>is not necessary</u> to prevent errors due to unexpected interactions between threads

Atomicity

[Flanaghan]

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

```
void deposit(int x){
synchronized(this) {
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

semantics:
lock acquire/release

```
void deposit(int x){
atomic {
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

semantics:
(behave as if)
no interleaved execution

No fancy hardware, code restrictions, deadlock, or unfair scheduling (e.g., disabling interrupts)

AtomJava

[Grossman]

- New prototype from the University of Washington
  - Based on source-to-source translation for Java
- Atomicity via locking (object ownership)
  - Poll for contention and rollback
  - No support for parallel readers yet
- Key pieces of the implementation
  - All writes logged when an atomic block is executed
  - If thread is pre-empted in atomic, rollback the thread
  - Duplicate so non-atomic code is not slowed by logging
  - Smooth interaction with GC