Thread basics

Threads and processes

- Every program you run starts a process
  - A *process* is the entity associated with a running program that owns the resources of the running program and that is scheduled and managed by the operating system.
  - A *process* has its own address space, open files, is allocated physical memory, etc.
- Every process has at least one *thread*
  - A thread has its own program counter and registers
  - System resources used by the thread are owned by the process
  - In particular, all threads associated with a process share the same address space.

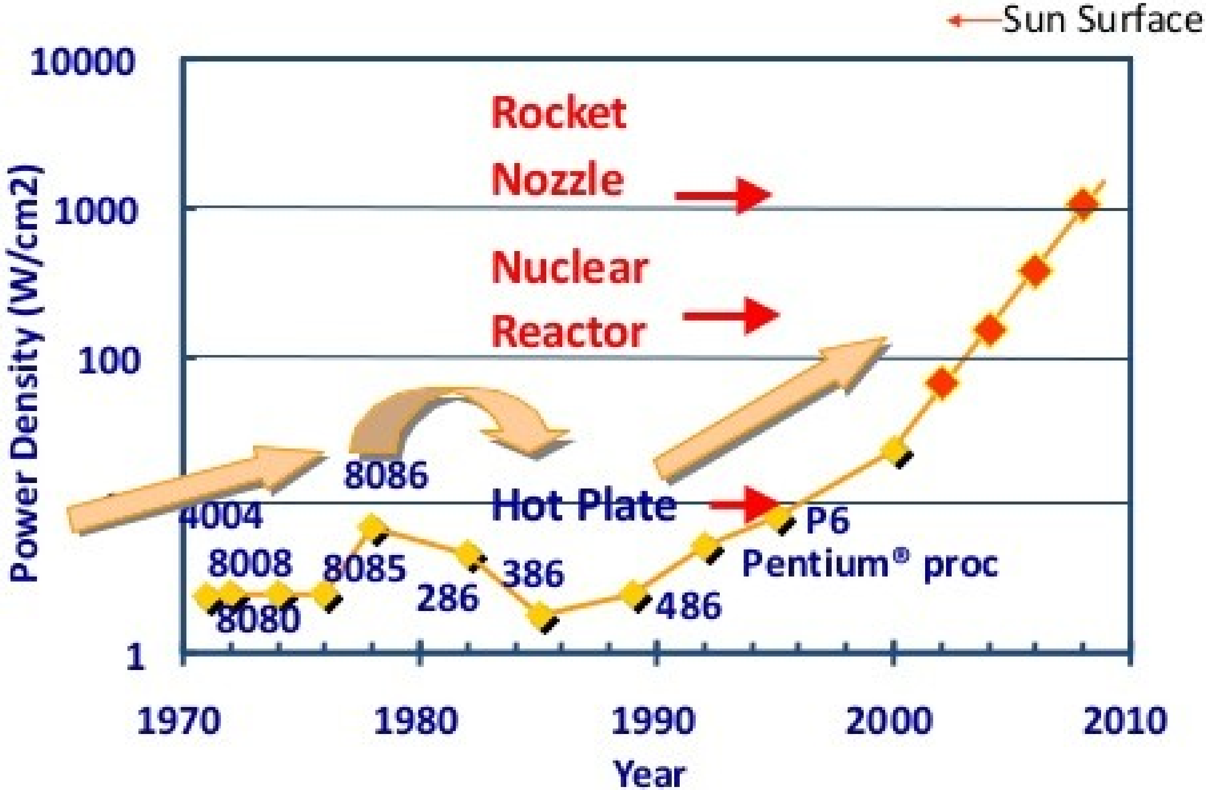
Why use threads?

- Easier programming
  - Many tasks whose execution needs to appear to be interleaved/happening at the same time
  - Some tasks can run forever (e.g., watch for mouse input)
  - Having a loop iterate over them *and* making sure each tasks gets its share of the processor can lead to complex programs
- Better performance
  - To use all of the cores in a multicore processor we need at least one thread for each core

Why multicores

- Life was simpler when processor clock rates doubled every couple of years or so
- Processors got faster, enabling more complicated software, when motivated faster processors (and buying a new machine) which motivated even more complicated software . . .
- *If something cannot go on forever, it will stop.* --Stein's Law, first pronounced in the 1980s by Herbert Stein
  - Always true of exponentials
  - $E = 1/2*C*V^2$, where E is energy, C is capacitance and V is voltage.
  - Higher frequencies require higher voltages
  - More cores increase C, which increases energy linearly

Power density



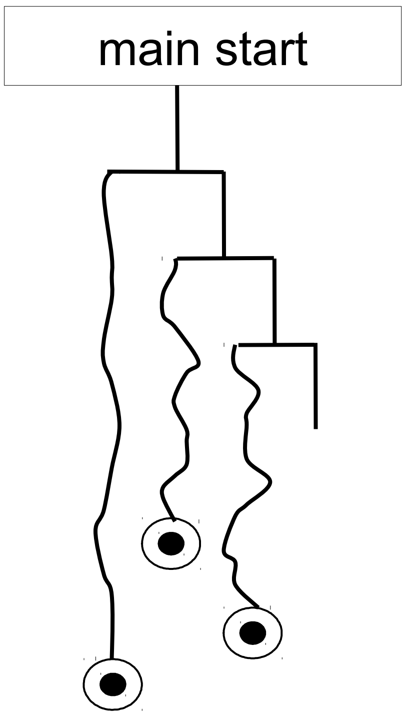
Power density too high to keep junctions at low temp

Courtesy, Intel

Java offers good support for multithreading

Java Thread

```
class YourClass extends Thread {
    public void run () {
        // code for the thread, i.e. what it does
    }
}
...
public static void main(String [] args) {
    YourClass t1 = new YourClass("...");
    t1.start();
}
```



Thread Execution Time

One core or processor

t1			t1			t1		
	t2				t2		t2	
		t3		t3				t3

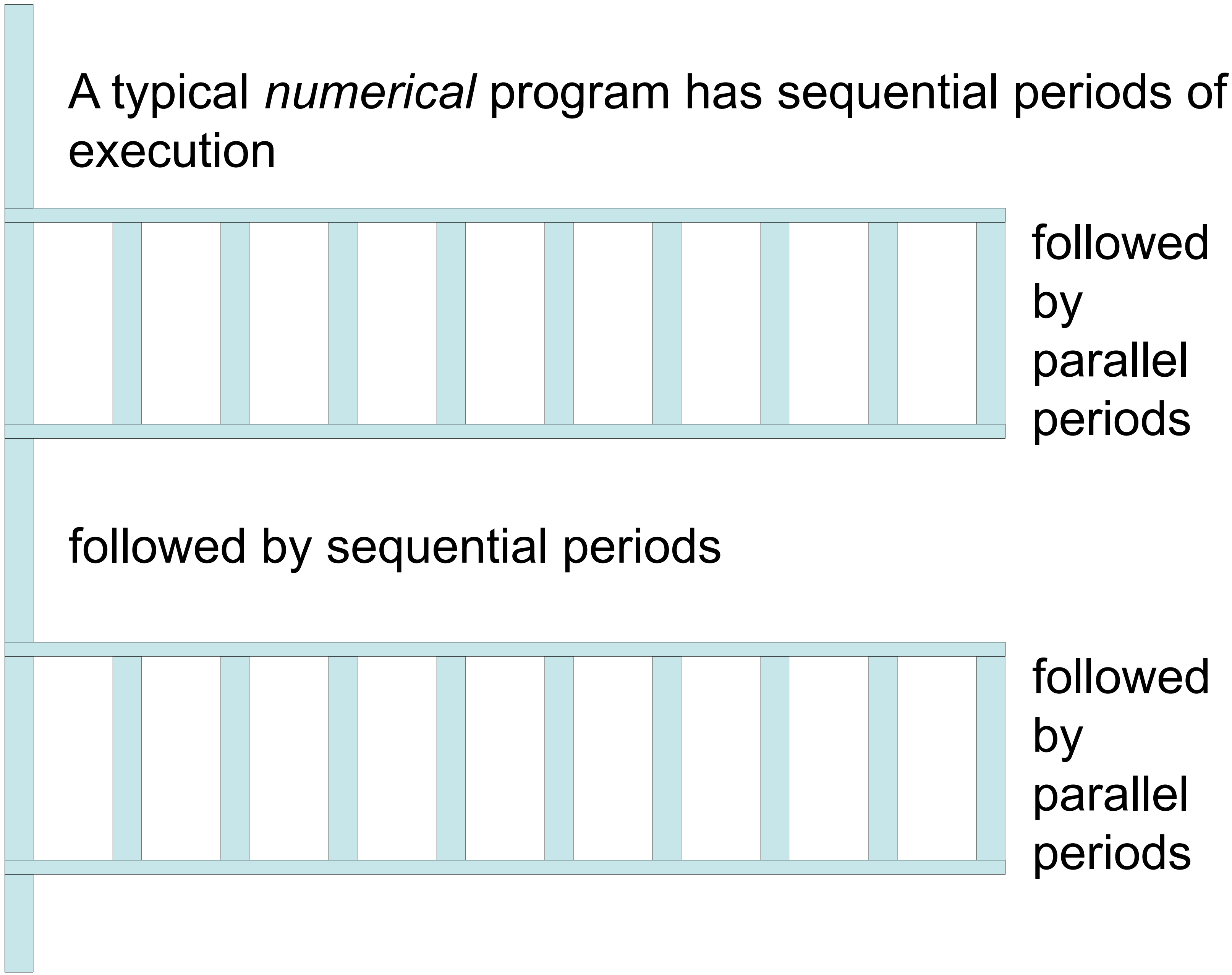
>= 2 cores or processors

t1	t1		t1	t1		t1	t1	
t2		t2		t2	t2		t2	t2
	t3	t3	t3		t3	t3		t3

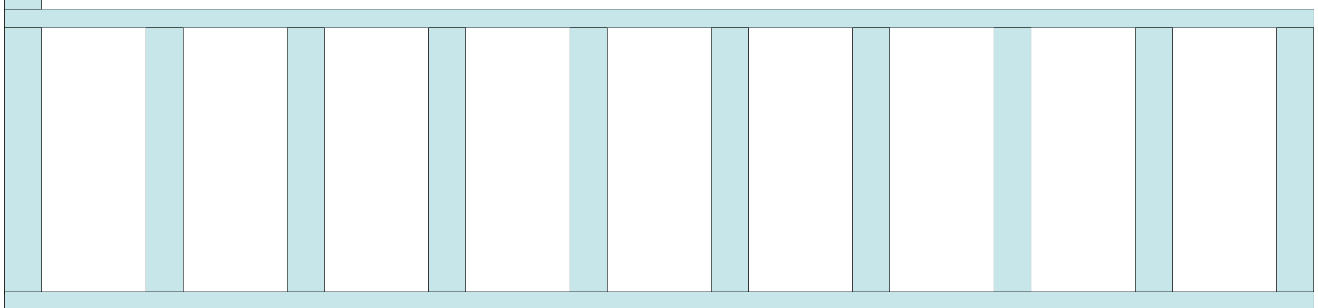
time

Many Threads, Few Processors

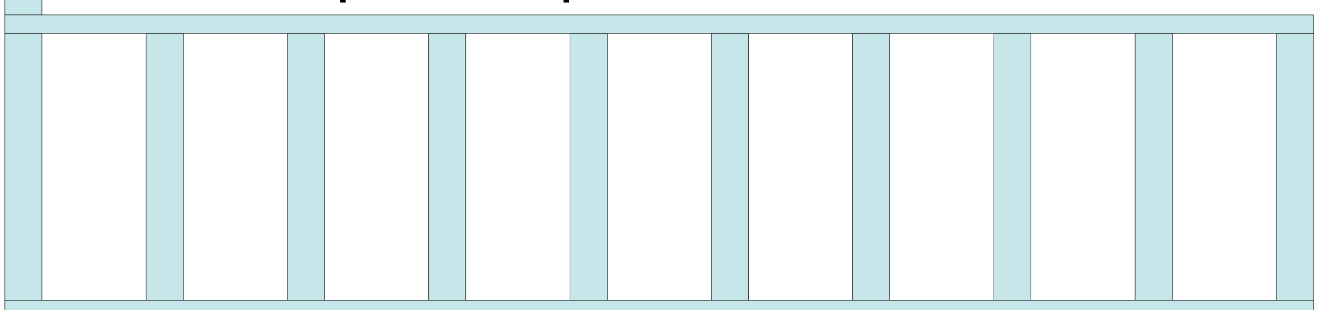
- advantages of many threads, even on single processor
  - impression of continuous progress in all threads
  - improve utilization of different components
  - handle user inputs more quickly
- disadvantage of many threads
  - add slight work to program structure
  - adds scheduling overhead
  - incur overhead in thread creation
  - cause complex interleaving the execution and possibly wrong results (if you do not think "in parallel")


A typical *numerical* program has sequential periods of execution
followed by parallel periods
followed by sequential periods
followed by parallel periods

As programmers, we can spawn new threads at the start of a parallel phase, and kill them at the end of the phase



Or we can start the threads once, and at the end of a parallel period put them into a pool to be reused at the next parallel period



Or have them suspend to begin working again

A Second Reason for Threads

- Let say you have a game that is handling multiple players and characters
- The game also needs to monitor keyboard input, mouse clicks, etc.
- There are several ways to code this
  - One big loop that goes over everything
  - A thread that monitors input and an action loop
  - A thread for input and each character

One big loop

```
while (true) {
   check if new input, if so, put on the input queue // what if
      // we need to pause to check what is coming next to
      // complete a command to put on the input queue?
   update char1 action
   update char2 action
   . . .
   update charn action
}
```

Using two threads (changed)

Thread 0:

Check for input, clean it up, put on an input queue

Thread 1:

```
while (true) {
  update char1 action
  update char2 action
  . . .
  update charn action
}
```

Using many threads (changed)

Thread 0:

Check for input, clean it up, put on an input queue

Thread 1:

char$_1$ actions

. . .

Thread n:

update char$_n$ action

}

Two ways to spawn threads in Java
First Way

```
public class myThread extends Thread {
   . . .
   public void run( ) {
      // thread actions here
   }
   . . .
}

myThread t1 = new Mythread(...);
t1.start( ); // indirectly invokes t1.run( )
```

- Inherit from the `Thread` class
- Invoke the **`run`** method on the object via the *start* method call
- We don't have to write start -- it comes for free (inherited from Thread)
- The `run` method can be viewed as the "`main`" method of the thread

Two ways to spawn threads in Java Second Way

```
public class myRunnable
   extends C implements runnable {
   public myRunnable( ) {
     // constructor stuff
   public void run( ) {
     // thread actions here
   }
   . . .
}
. . . .
Thread t1 = new Thread(new MyRunnable( )).start( );
```

- Implement `Runnable` interface
- Invoke the `start` method on the `Thread` object
- The `start` method calls the `run` method after some underlying system actions.

From a discussion on StackOverflow

http://stackoverflow.com/questions/541487/implements-runnable-vs-extends-thread

Moral of the story:

Inherit only if you want to override some behavior.

Or rather it should be read as:

Inherit less, interface more.

Or, in other words, implementing *Runnable* is preferable to *extends Thread*

Calling run directly does not start a new thread

The difference between run and start

Sai Hegde at http://www.coderanch.com/t/234040/threads/java/Difference-between-run-start-method

```
1.Thread t1 = new Thread();
2.Thread t2 = new Thread();
3.t1.run();
4.t2.run();
```

t1.run( ) is guaranteed to completely execute before *t2.run*, i.e. it does not execute the two *run* calls asynchronously with the calling code. The *run* method is executed with the same thread that calls *t1.run( )* and *t1.run( )*.

Often not useful.

Calling *run* does not *start* a new Thread

```
1.Thread t1 = new Thread();
2.Thread t2 = new Thread();
3.t1.run();
4.t2.run();
```

t1.run and t2.run will execute one after the other like any other method call

Calling *start* does start a new *Thread*

```
1.Thread t1 = new Thread();
2.Thread t2 = new Thread();
3.t1.start();
4.t2.start();
```

t1.start and t2.start can, and usually will, execute asynchronously with respect to one another *and* the calling thread.

An attempt at a humorous example of this

```
private class LawnMower extends Thread {
    public void run() {
        cutTheGrass();
    }
}
public void doChoresFirstThenReadComics() {
    new LawnMower.run();
    readComics();
}
public void readComicsWhileSomeoneElseDoesChores() {
    new LawnMower.start();
    readComics();
}
```

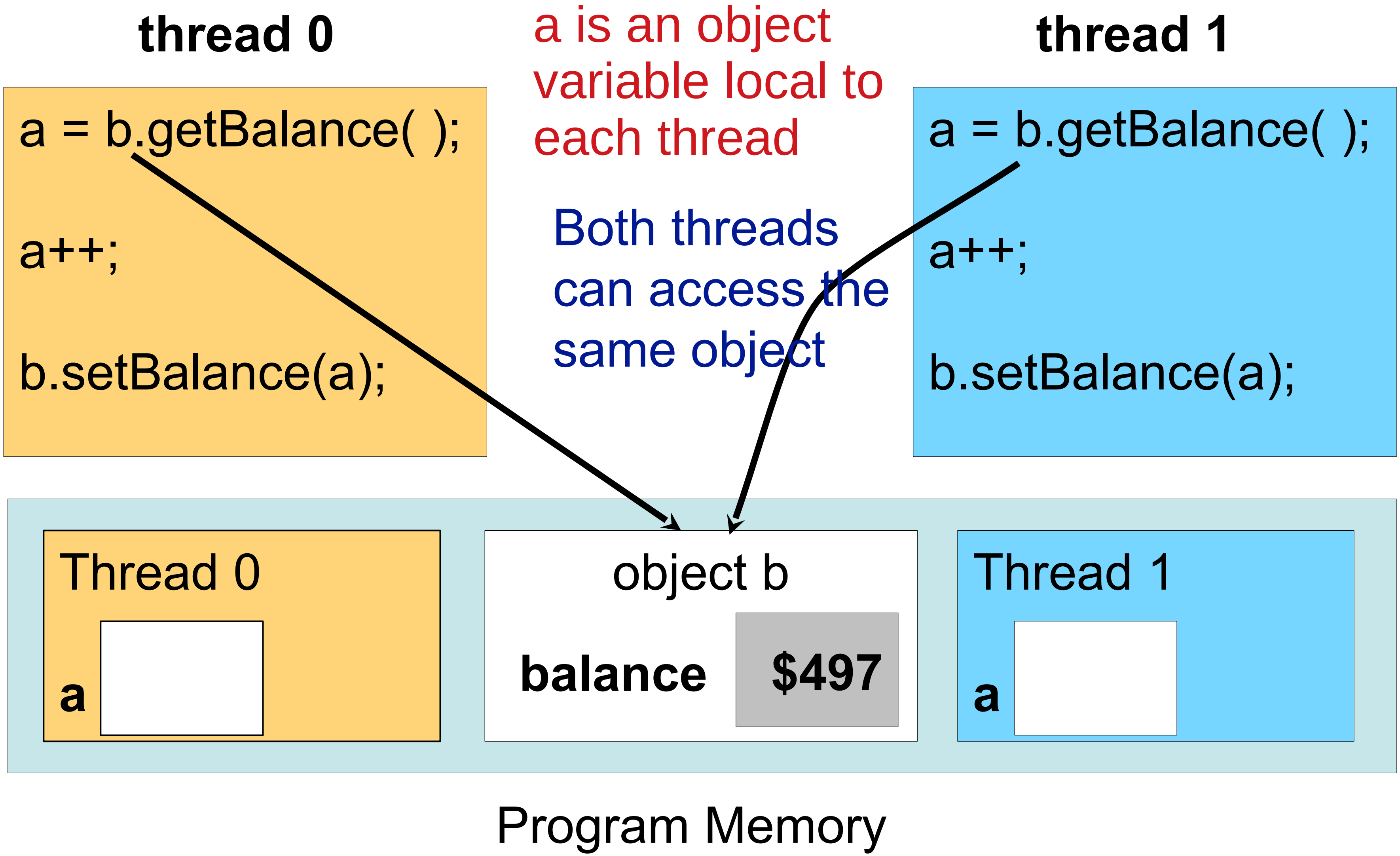
What happens with *start*?

- *Thread t1 = new Thread( )* creates a new Java *Thread object*.
- *t1.run( )* invokes the run method on that object
- To get asynchronous execution, a new *thread*, i.e., a new locus of control, needs to be created.
- This is what *start* does.
  - When start is executed, it creates a new system *thread* (generally an OS thread on most implementations)
  - it executes the run method in that new thread.
  - This is what lets the actions performed by the run method execute asynchronously with other code.

All threads for a given process share memory

- If thread T0 writes a value to X, and thread T1 reads X, the value of X will (eventually) change
- The major challenge of writing correct multi-threaded programs is managing accesses to variable shared across different threads

ordering and *atomicity* are important and different



thread 0

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

thread 1

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

Program Memory

thread 0

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

thread 1

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

Program Memory

thread 0

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

thread 1

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

thread 0	object b	thread 1
a **$498**	**balance** **$498**	**a** **$497**

Program Memory

thread 0

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

The end result probably should have been $499. One update is lost.

thread 1

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

Program Memory

synchronization enforces atomicity

thread 0

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

Make them *atomic* using synchronized

thread 1

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

thread 0

a

object b

balance **$497**

thread 1

a

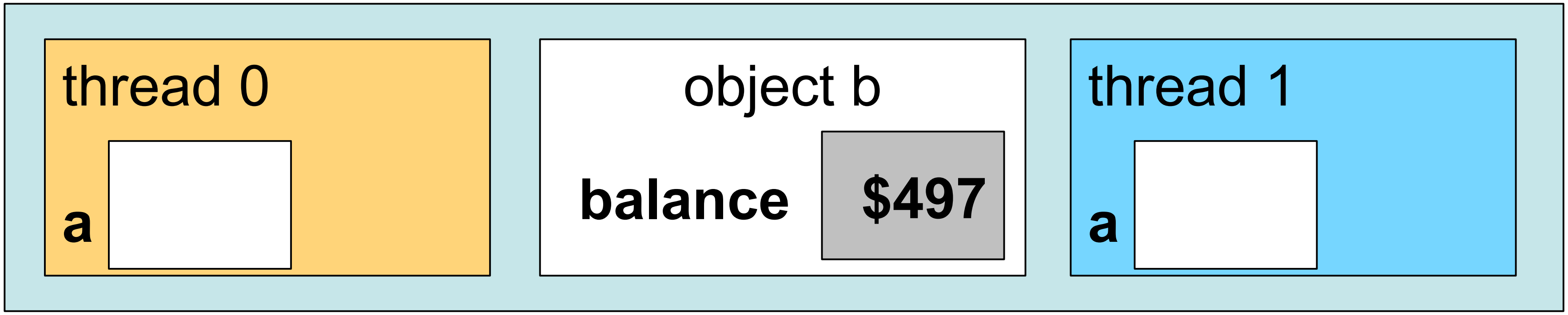
Program Memory

One thread acquires the lock

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free

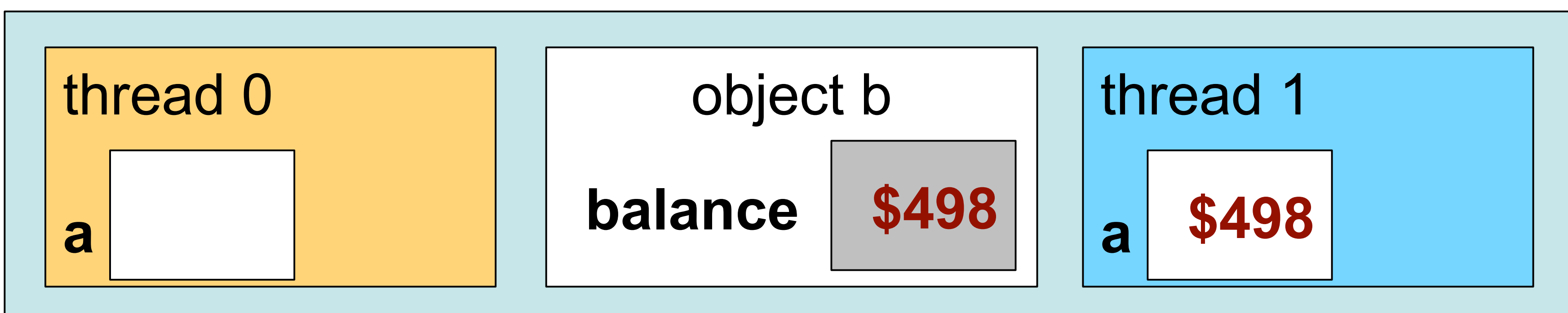


One thread acquires the lock

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free

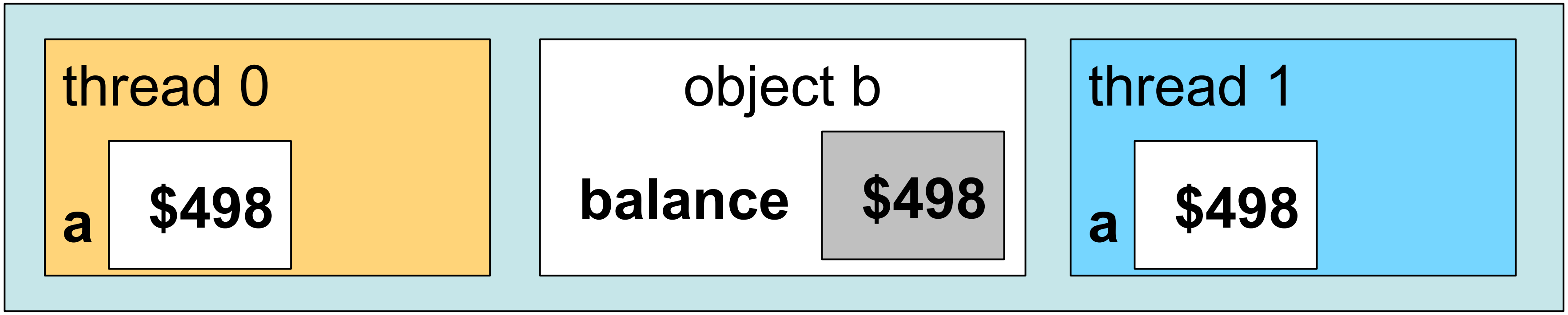


One thread acquires the lock

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free

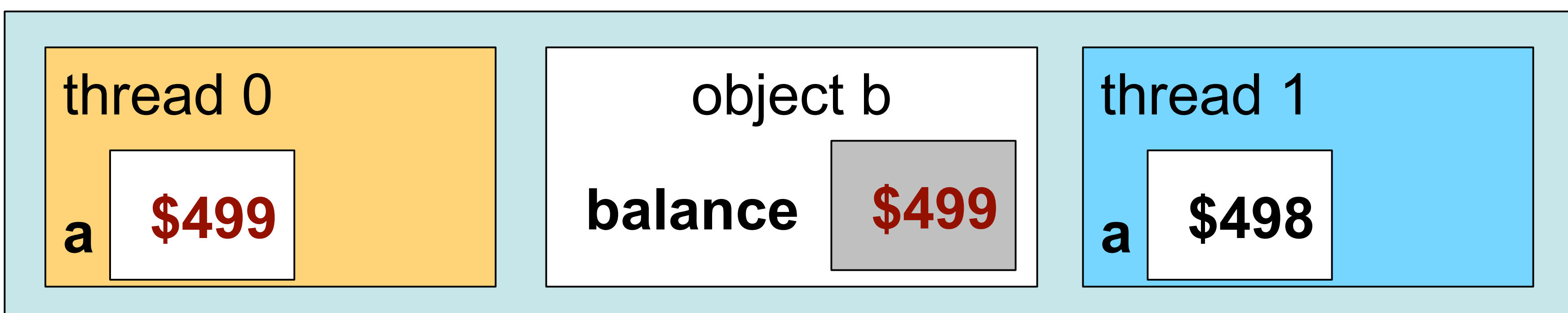


One thread acquires the lock

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free



Locks typically do not enforce ordering

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

Either order is possible

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

For many (but not all) programs, either order is correct

Java Locks

- Every object can be locked
- the code `synchronized(b) {stmt_list}` says that no other code synchronized on *the object referenced by* **b** can execute at the same time as `stmt_list` in the thread holding the lock.
- By locking on objects accessed in a block of code, the operations can be made *atomic*. Assume the code accesses objects o1 and o2:
  - Any other code accessing o1 and o2 has to synchronize on at least one lock that is the same
  - Simply getting a lock does not make the code atomic: *it is necessary for other code to cooperate and try and get at least one lock that is the same*
  - This violates encapsulation, but life is tough

```
synchronized(o1) {    synchronized(o2) {
    o1.foo( );            o1.foo( );
    o2.bar( );            o2.bar( );
}                     }
```

Wrong - no synchronization assuming o1, o2 different objects

```
synchronized(o1) {    synchronized(o1) {
    o1.foo( );            o1.foo( );
    o2.bar( );            o2.bar( );
}                     }
```

Works – synchronized assuming o1 is same in both calls

```
synchronized(o2) {    synchronized(o2) {
    o1.foo( );            o1.foo( );
    o2.bar( );            o2.bar( );
}                     }
```

Works – synchronized assuming o1 is same in both calls

```
synchronized(o3) {
    o1.foo( );
    o2.bar( );
}
```

```
synchronized(o3) {
    o1.foo( );
    o2.bar( );
}
```

Works

```
synchronized(o1) {
    synchronized(o2)
    o1.foo( );
    o2.bar( );
}
```

```
synchronized(o1) {
    synchronized(o2)
    o1.foo( );
    o2.bar( );
}
```

Works

```
synchronized(o2) {
    synchronized(o1)
    o1.foo( );
    o2.bar( );
}
```

```
synchronized(o2) {
    synchronized(o1)
    o1.foo( );
    o2.bar( );
}
```

Works

Acquiring multiple locks can lead to deadlock

```
synchronized(o1) {
    synchronized(o2)
    o1.foo( );
    o2.bar( );
}
```

```
synchronized(o2) {
    synchronized(o1)
    o1.foo( );
    o2.bar( );
}
```

very dangerous

When doing multithreaded programming, assume that anything bad that can happen will happen if it is not prevented from happening by locks or other mechanisms.

Bugs involving races, deadlock, etc. are incredibly hard to find because the program behavior is non-deterministic.

Can lead to deadlock

```
synchronized(o1) {
    synchronized(o2)
    o1.foo( );
    o2.bar( );
}
```

```
synchronized(o2) {
    synchronized(o1)
    o1.foo( );
    o2.bar( );
}
```

very dangerous

Can lead to deadlock

```
synchronized(o1) {
    synchronized(o2)
    o1.foo( );
    o2.bar( );
}
```

```
synchronized(o2) {
    synchronized(o1)
    o1.foo( );
    o2.bar( );
}
```

very dangerous

o1 lock

The left thread cannot get `o2`'s lock, the right thread cannot get `o1`'s lock, so neither thread can finish and release their locks -- *deadlock*!

o2 lock

Can lead to deadlock

```
synchronized(o1) {
    synchronized(o2)
    o1.foo( );
    o2.bar( );
}
```

```
synchronized(o2) {
    synchronized(o1)
    o1.foo( );
    o2.bar( );
}
```

very dangerous

o1 lock

There is always an ordering cycle in programs that can deadlock.

o2 lock

Synchronized methods

```
Class B {
    . . .
    synchronized void foo(T o1) {
        o1.foo( );
        o2.bar( );
    }
    . . .
}
. . .
B b = new B( );
b.foo(ox);
```

When `foo` is invoked, a lock is acquired on object ref'd by **`b`**, not **`ox`**

Synchronized methods - how this works

```
Class B {
    . . .
    synchronized void foo(T o1, B this) {
        o1.foo( );
        o2.bar( );
    }
    . . .
}
. . .
B b = new B( );
b.foo(ox, b);
```

When `foo` is invoked, a lock is acquired on object ref'd by **`b`**, not **`ox`**

Synchronized method semantics

```
Class B {
   . . .
   synchronized void foo(T o1) {
      o1.foo( );
      o2.bar( );
   }
```

As if a lock is acquired on the *this*, i.e. *synchronized(this)* within the method.

```
Class B {
   . . .
   void foo(T o1) {
      synchronized(this) {
         o1.foo( );
         o2.bar( );
      }
   }
```

Synchronized methods

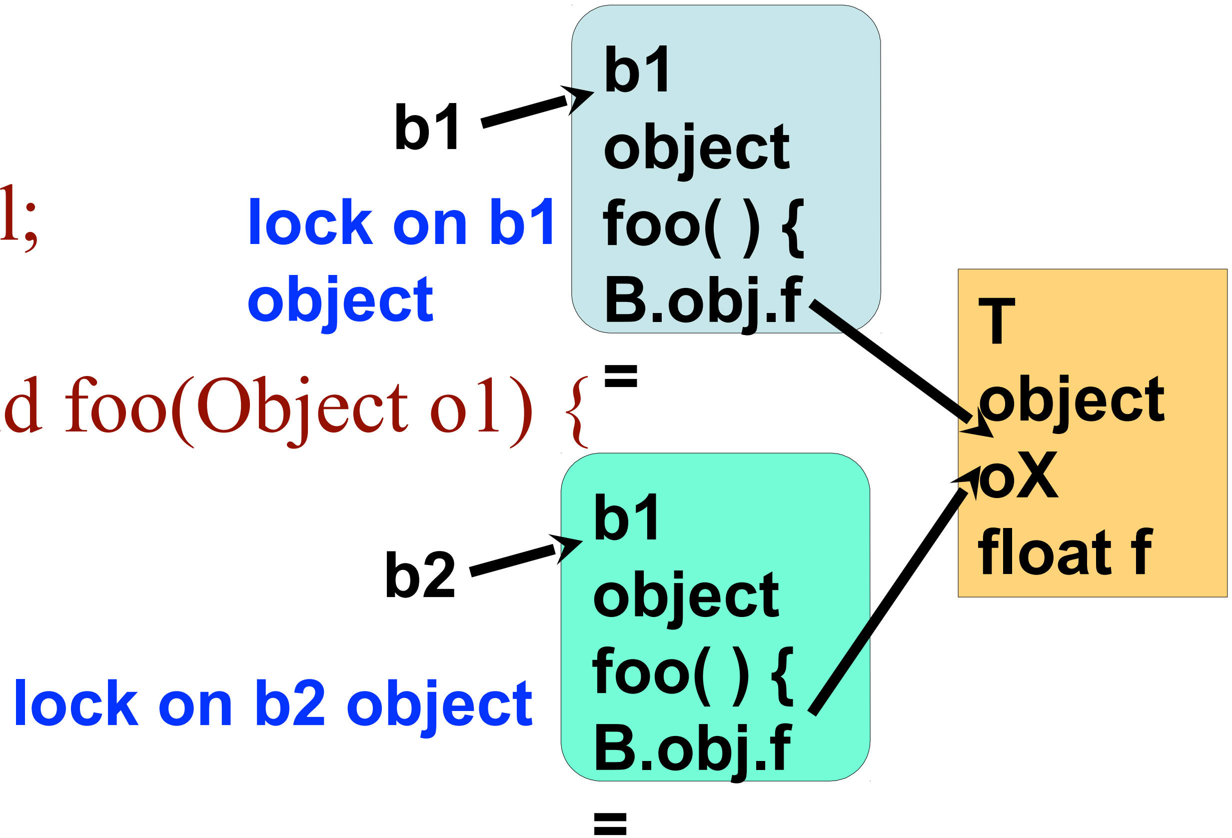
```
Class B {
  static T obj = null;
  B(T t) {obj = t;}
  synchronized void foo(Object o1) {
    B.obj.f = . . .
  }
}
```

There will be a *race* on the access to B.obj.f (i.e. oX.f) in the calls to b1.foo and b2.foo.

Thread 0	Thread 1
B b1 = new B(oX);	B b2 = new B(oX);
b1.foo( );	b2.foo( );

Synchronized methods

```
Class B {
  static T obj = null;
  B(T t) {obj = t;}
  synchronized void foo(Object o1) {
    B.obj.f = . . .
  }
}
```



Thread 0

```
B b1 = new B(oX);
b1.foo( );
```

Thread 1

```
B b2 = new B(oX);
b2.foo( );
```

In this case synchronize on B.obj

```
Class B {
  T obj = null;
  B(T t) {obj = t;}
   synchronized void foo(Object o1) {
      synchronize(obj) {
         obj.f = . . .
      }
   }
}
```

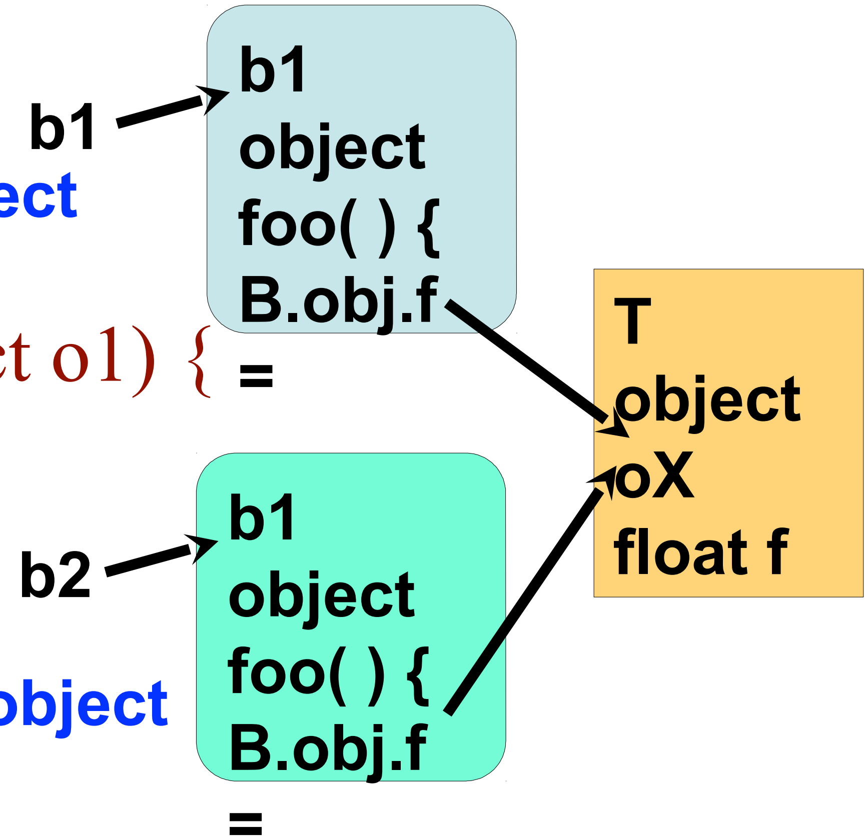
Both threads now synchronize on the field being accessed

Thread 0	Thread 1
B b1 = new B(oX);	B b2 = new B(oX);
b1.foo( );	b2.foo( );

Synchronized methods

```
Class B {
  T obj = null;
  B(T t) {obj = t;}
   synchronized void foo(Object o1) {
     synchronize(obj) {
       obj.f = . . .
     }
   }
}
```

lock on obj object



lock on obj object

Thread 0

```
B b1 = new B(oX);
b1.foo( );
```

Thread 1

```
B b2 = new B(oX);
b2.foo( );
```

A general rule

- To avoid races do one of the following
  - Always synchronize on the shared object
  - Always synchronize on another object that is used everywhere in the program to synchronize the shared object(s) -- requires communication among the developers of the other parts of the program.

A general rule - first case

- To avoid races do one of the following
  - Always synchronize on the shared object(s)
  - Always synchronize on another object that is used everywhere in the program to synchronize the shared object

See the class B example three slides back

Be careful about deadlock!

A general rule - second case

- To avoid races do one of the following
  - Always synchronize on the shared object
  - Always synchronize on another object that is used everywhere in the program to synchronize the shared object

We will consider the code below

```
synchronized(o1) {
    synchronized(o2)
    o1.foo( );
    o2.bar( );
}
```

```
synchronized(o2) {
    synchronized(o1)
    o1.foo( );
    o2.bar( );
}
```

An example of the second case

```
synchronized(o1) {        synchronized(o2) {
    synchronized(o2)          synchronized(o1)
    o1.foo( );                o1.foo( );
    o2.bar( );                o2.bar( );
}                         }
```

```
class Lock {
    static l1 = new Object( );
}
```

```
synchronized(Lock.l1) {   synchronized(Lock.l1) {
    o1.foo( );                o1.foo( );
    o2.bar( );                o2.bar( );
}                         }
```

Question: Why not always use a single lock to synchronize everything?

Weird things happen without proper synchronization

What is the purpose of this code?
What value(s) can be printed for v1?
Does the while loop end?

Executes before threads are spawned

newVal = 0;
flag = 0

Thread 0

C.newVal = 52;

C.flag = 1;

Thread 1

while (C.flag == 0);

v1 = C.newVal;

System.out.println(v1);

Weird things happen without proper synchronization

Executes before threads are spawned

newVal = 0;
flag = 0

The bold line orders (and the dotted transitive order) are *NOT* guaranteed by Java and most languages. E.g., this would be an is an undefined C++ program.

Thread 0

C.newVal = 52;
C.flag = 1;

Thread 1

while (C.flag == 0);
v1 = C.newVal;
System.out.println(v1);

Weird things happen without proper synchronization

Executes before threads are spawned

newVal = 0;
flag = 0

No guarantee the **while** will ever end
No guarantee v1 will get the value assigned in Thread 0.

Thread 0

C.newVal = 52;

C.flag = 1;

Thread 1

while (C.flag = 0);

v1 = C.newVal;

System.out.println(v1);

What causes the problem?

Executes before threads are spawned

newVal = 0;
flag = 0

Compiler, processor or memory subsystem may reorder instructions. Register allocation may keep value of *flag* in the *while* loop in a register.

Thread 1

load C.flag, r1
while (r1 == 0);

v1 = C.newVal;

System.out.println(v1);

Thread 0

C.flag = 1;

C.newVal = 52;

Executes before threads are spawned

newVal = 0;
flag = 0

obj must be a reference to the same object in all threads

Thread 0

```
synchronized(obj) {
   C.newVal = 52;
   C.flag = 1;
}
```

Thread 1

```
synchronize(obj) {f = C.flag;}
while (f == 0) {
   synchronize(obj) {f = C.flag;}
}

v1 = C.newVal;

System.out.println(v1);
```

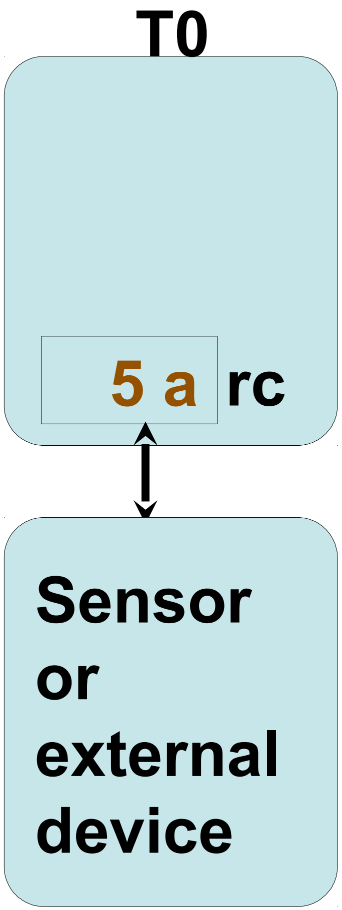
Synchronized also makes sure values are updated

- Compilers attempt to store values in registers
- Even if the cache entry for a variable is invalidated, the old or stale value may remain in a register
- When encountering a synchronized block in thread T, Java makes sure that
  - Values in registers accessed by T's code are refreshed (reload the registers from memory/cache)
  - ~~Reads and~~ Writes to memory initiated by thread T prior to the synchronized block are finished
- Before leaving a synchronized block Java makes sure that
  - all ~~reads and~~ writes have finished, i.e., the value are in memory.

Thus, synchronization does three things

- It enforces atomicity by letting the programmer only allow one thread at a time to access storage locations inside of synchronized code
- It forces the compiler to get fresh values for variables stored in registers
- It forces the compiler to write updated values to global memory

Volatile variables



- In embedded devices and controllers it is common to have a sensor/external device automatically update registers on the processors
- Program variables that contain values from this register should be updated every time they are read
- Volatile variables in Java can also be used to force threads to update values and write values within a synchronized block
- Use of volatile can decrease performance

Even long data types require attention

Not all primitive stores are atomic

```
public class C {
    static long li
= 0;
}
```

Thread 0

```
...
C.li = Long.MAX_VALUE( );
```

Thread 1

```
...
C.li = 0;
```

What are the allowed values for *C.li* after both stores (assuming they are unsynchronize)?

Not all primitive stores are atomic

Thread 0	Thread 1
...	...
C.li = Long.MAX_VALUE( );	C.li = 0;

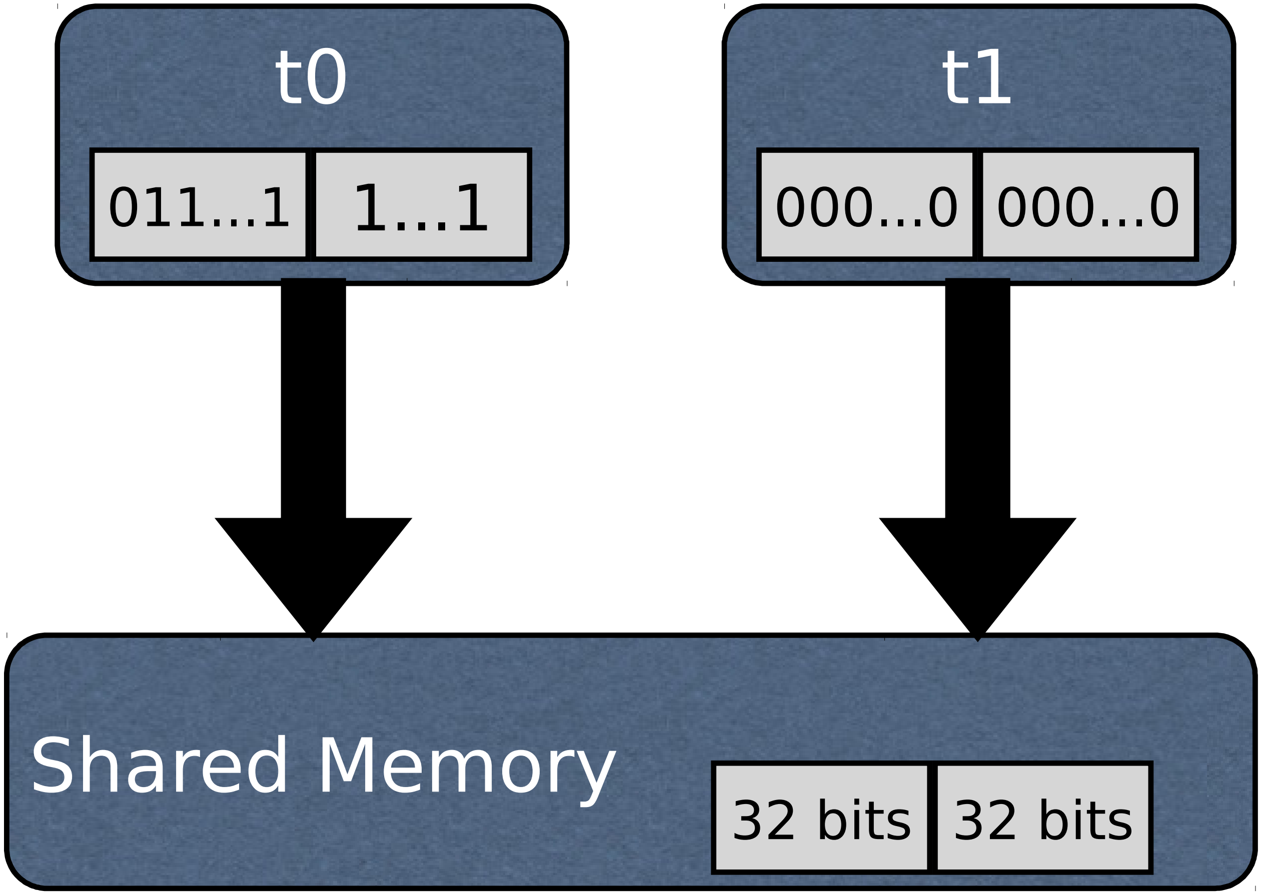
four values possible:

MAX_VALUE, 0,

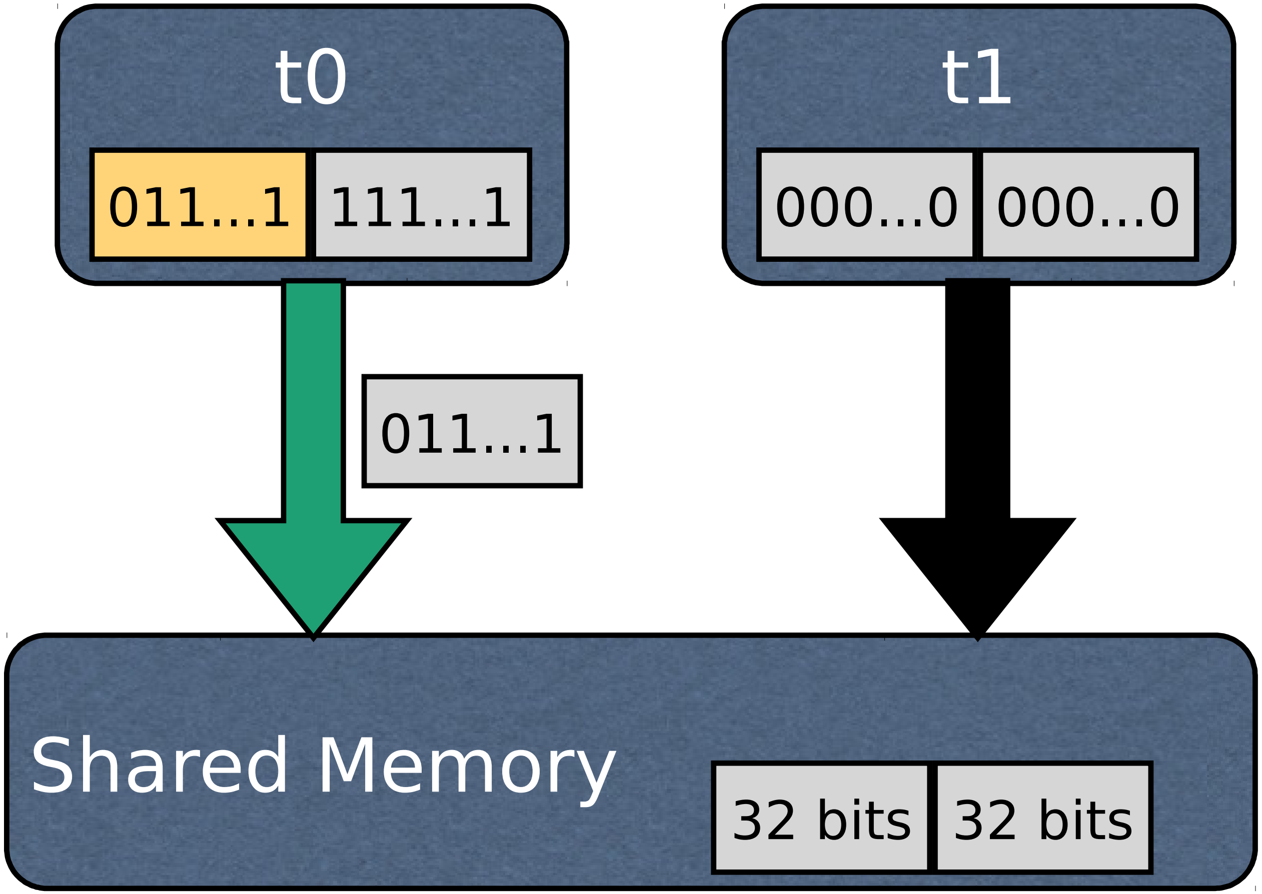
MAX_VALUE & 32(1).32(0) (32 1 bits followed by 32 0 bits)

MAX_VALUE & 32(0).32(1) (32 0 bits followed by 32 1 bits)

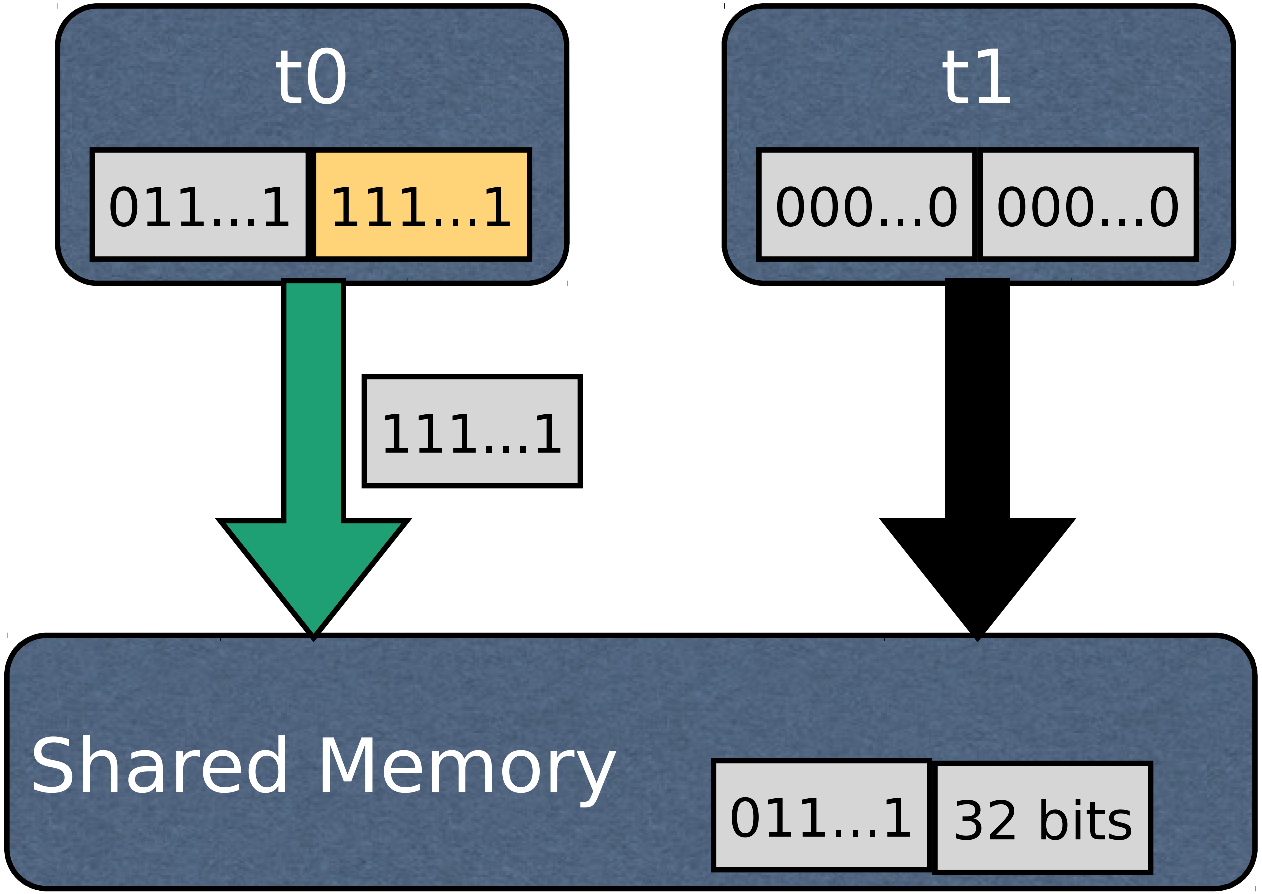
Why have such an abomination?



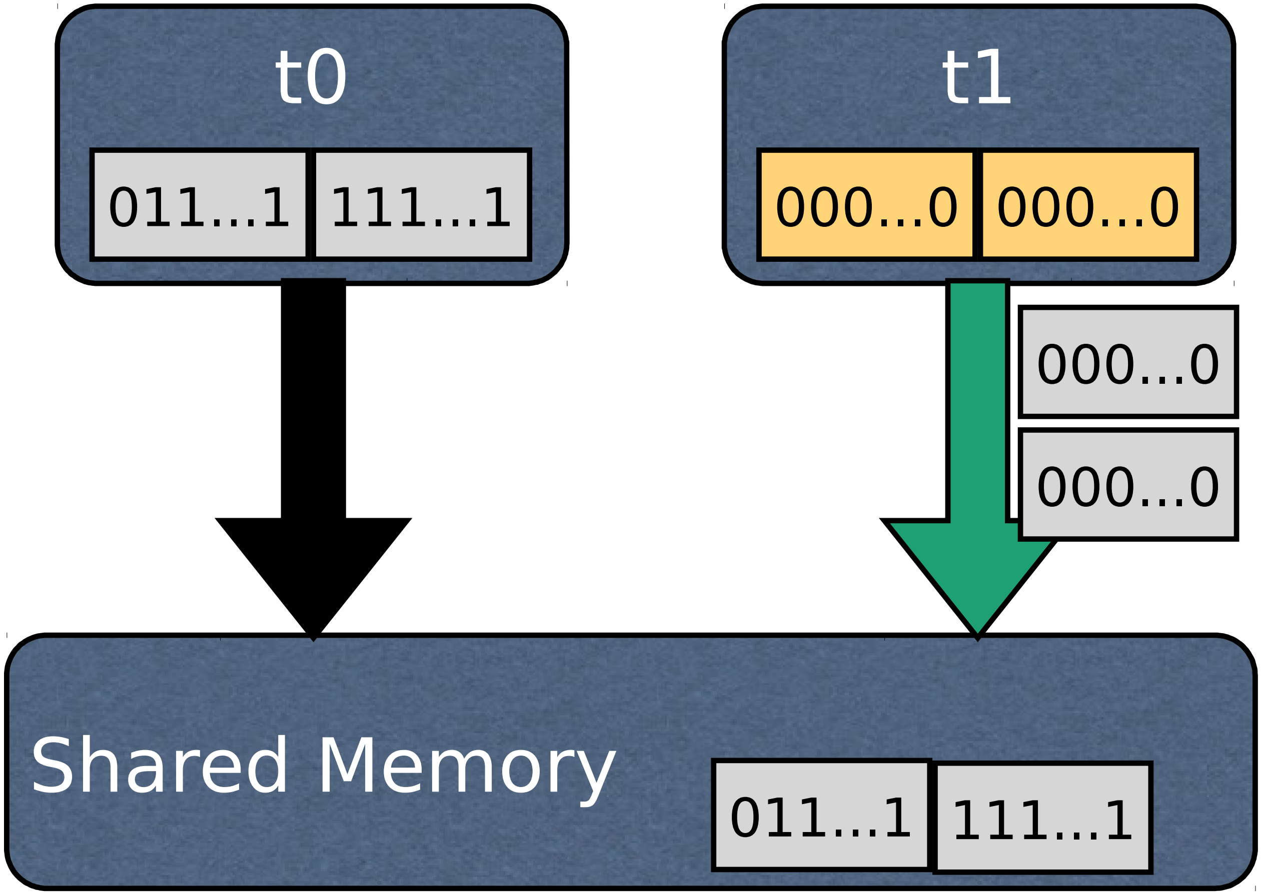
The right thing happens



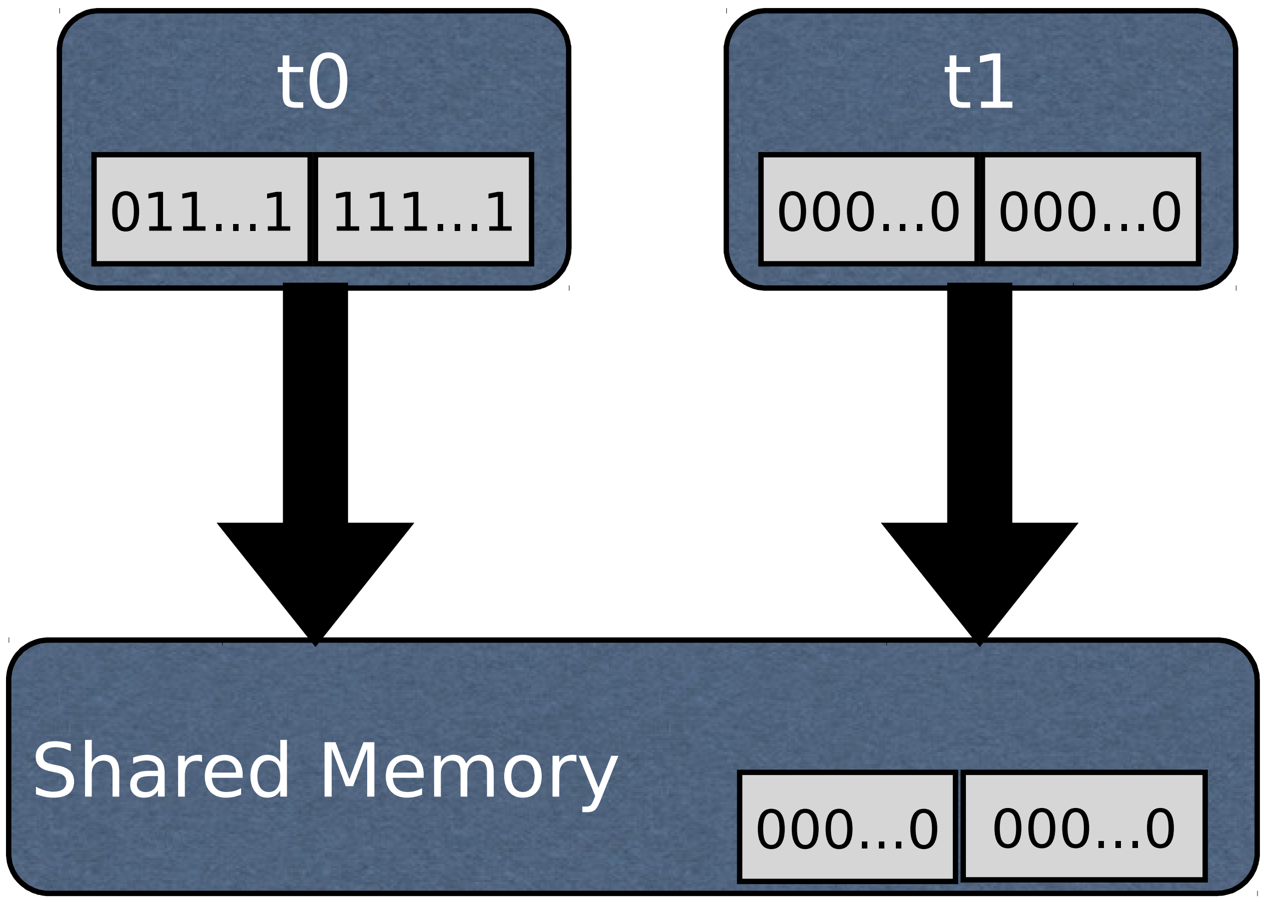
The right thing happens



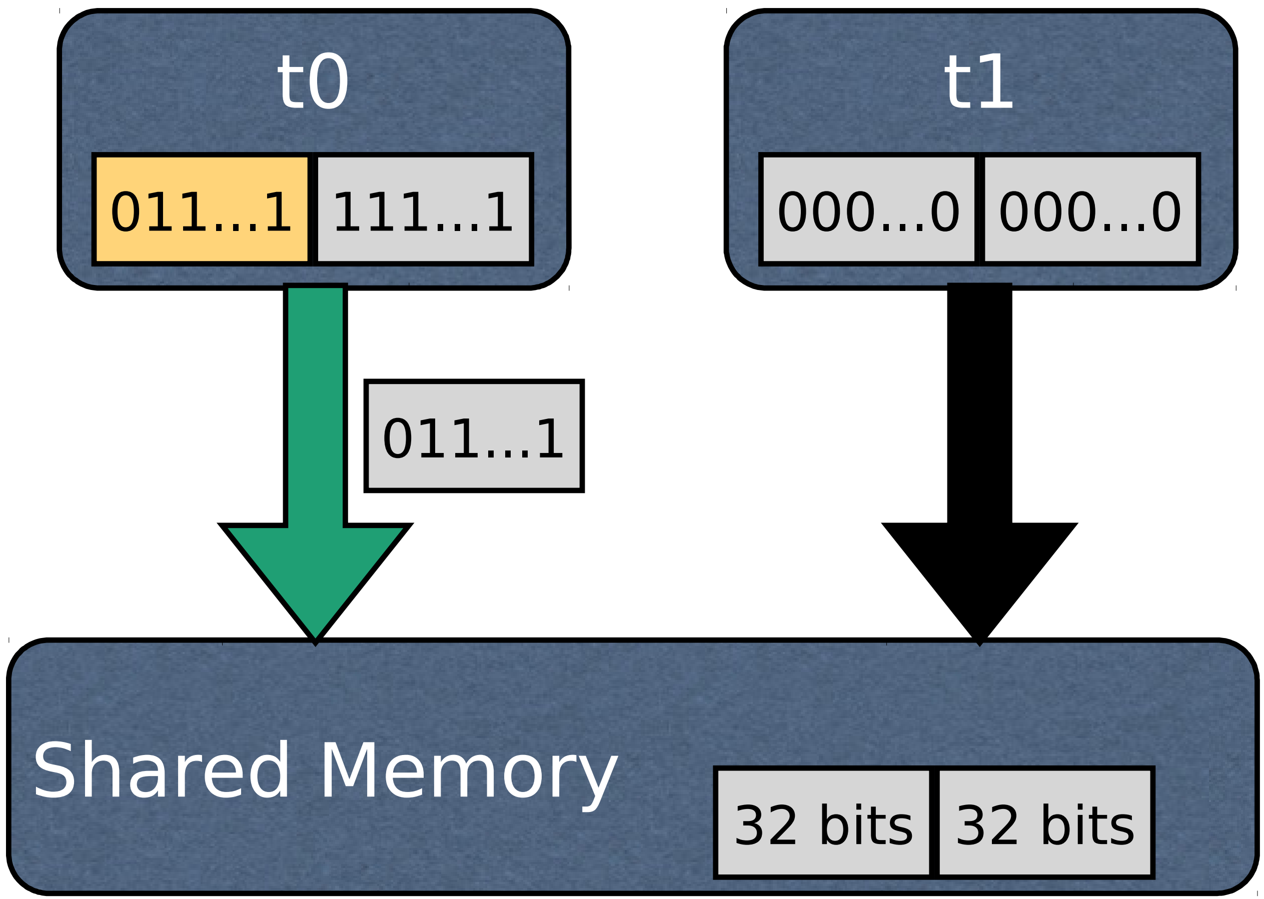
The right thing happens



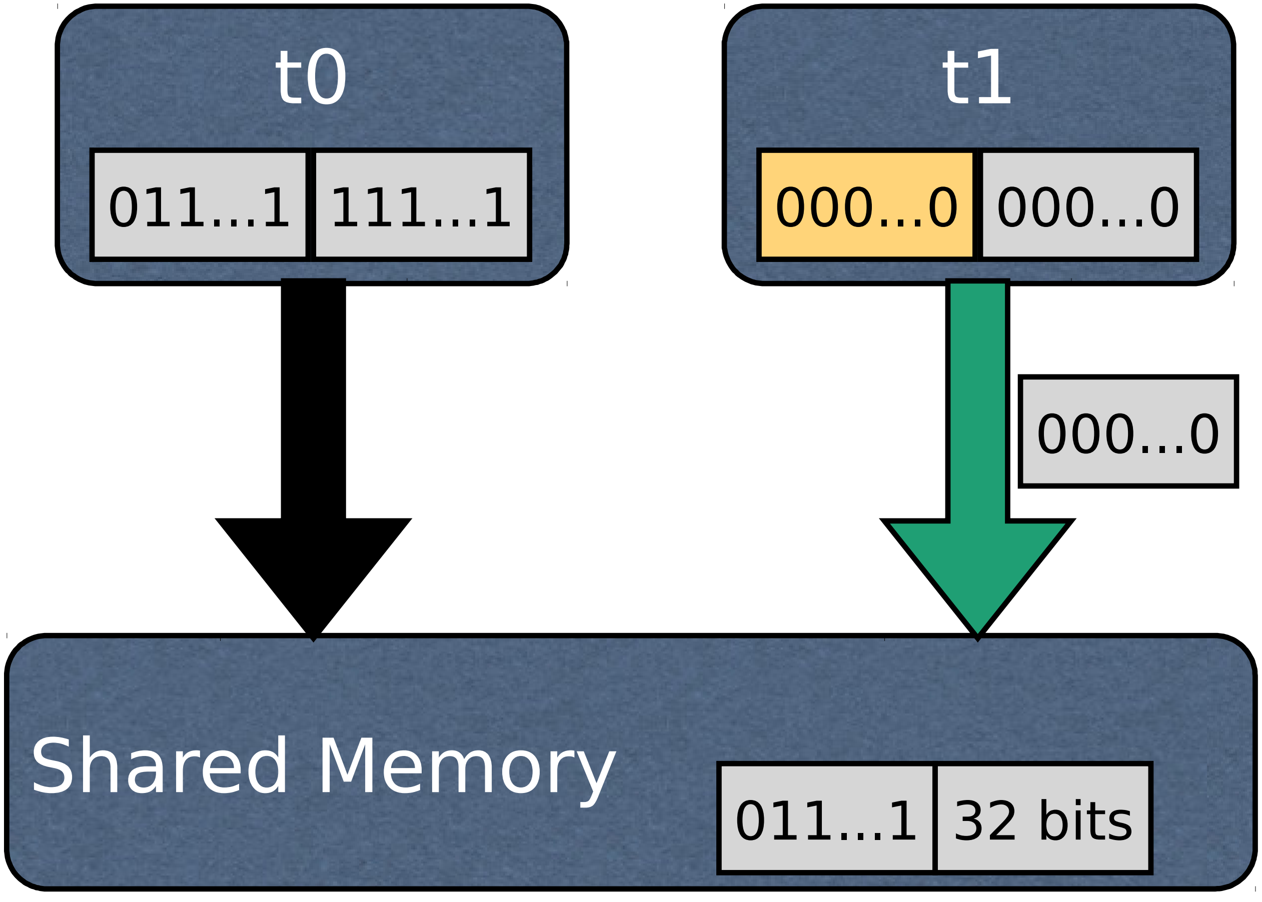
The right thing happens



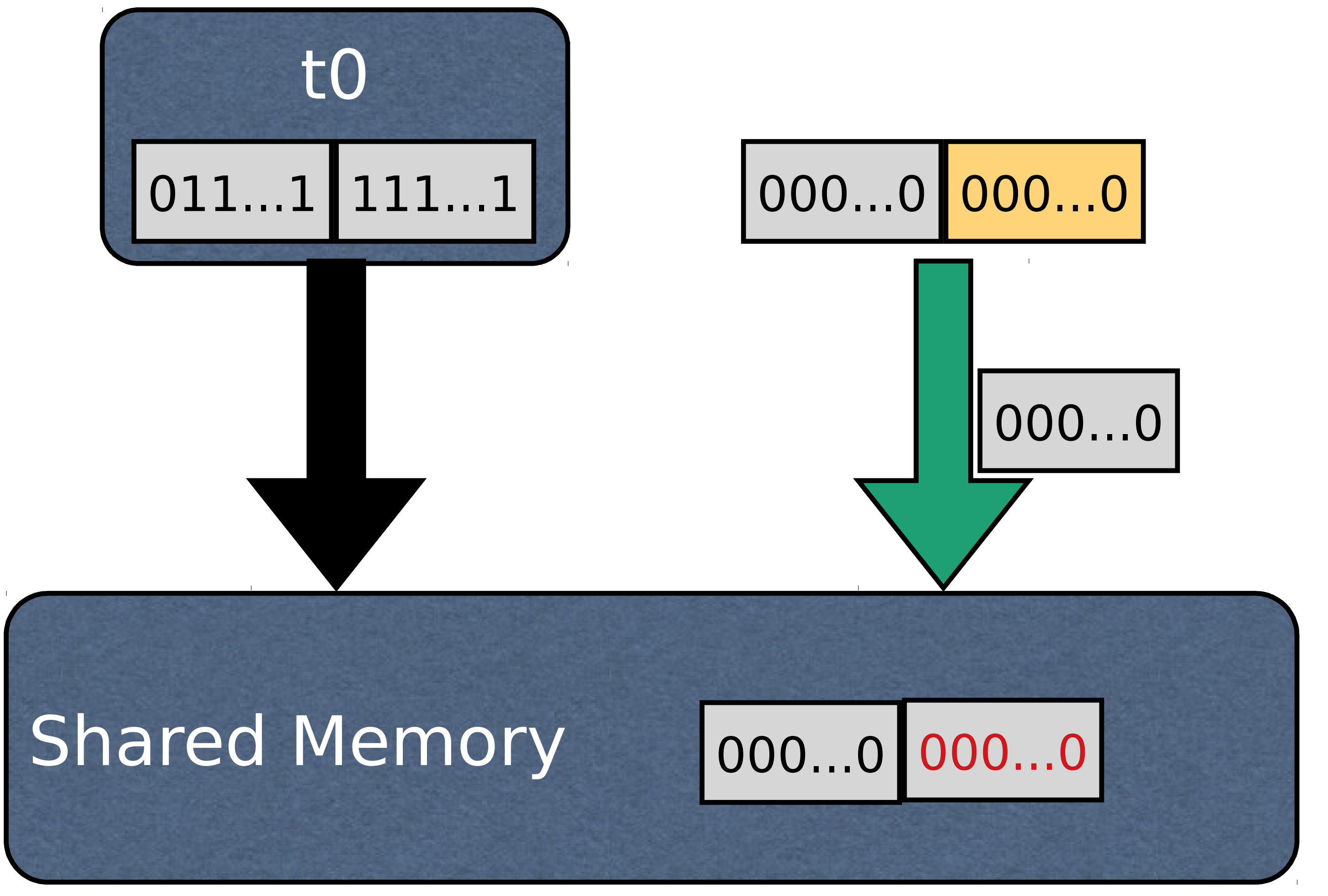
The wrong thing happens



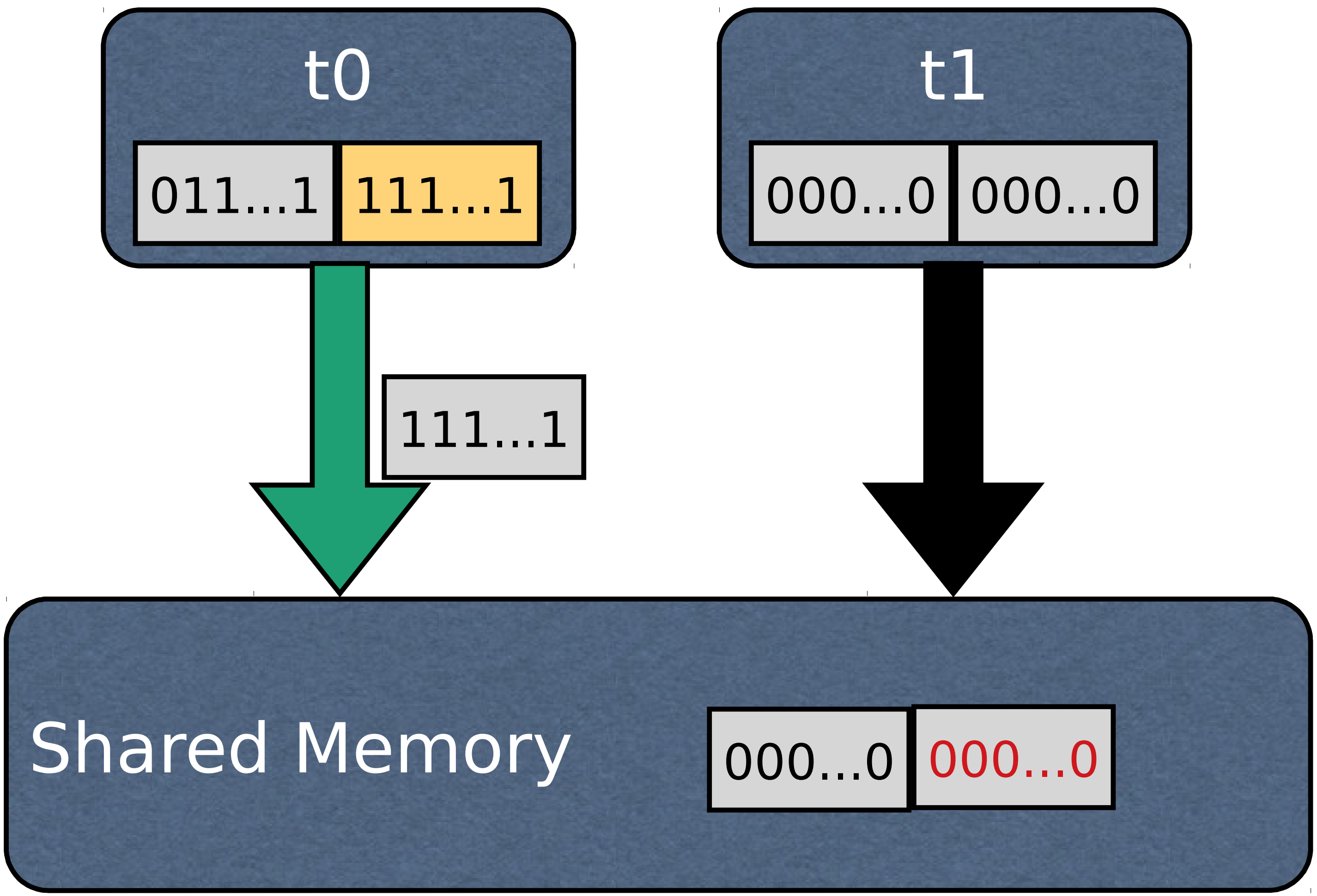
The wrong thing happens



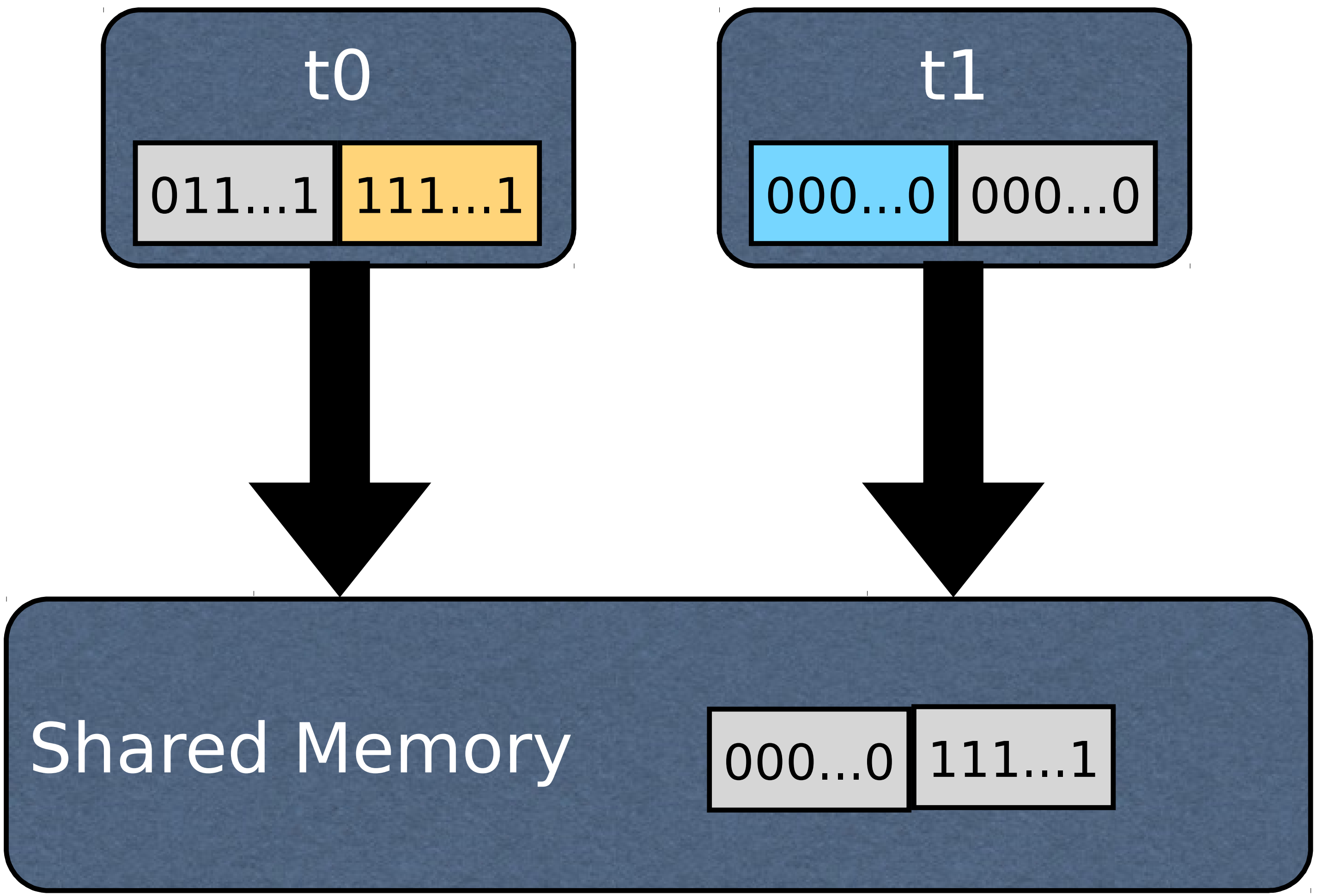
The wrong thing happens



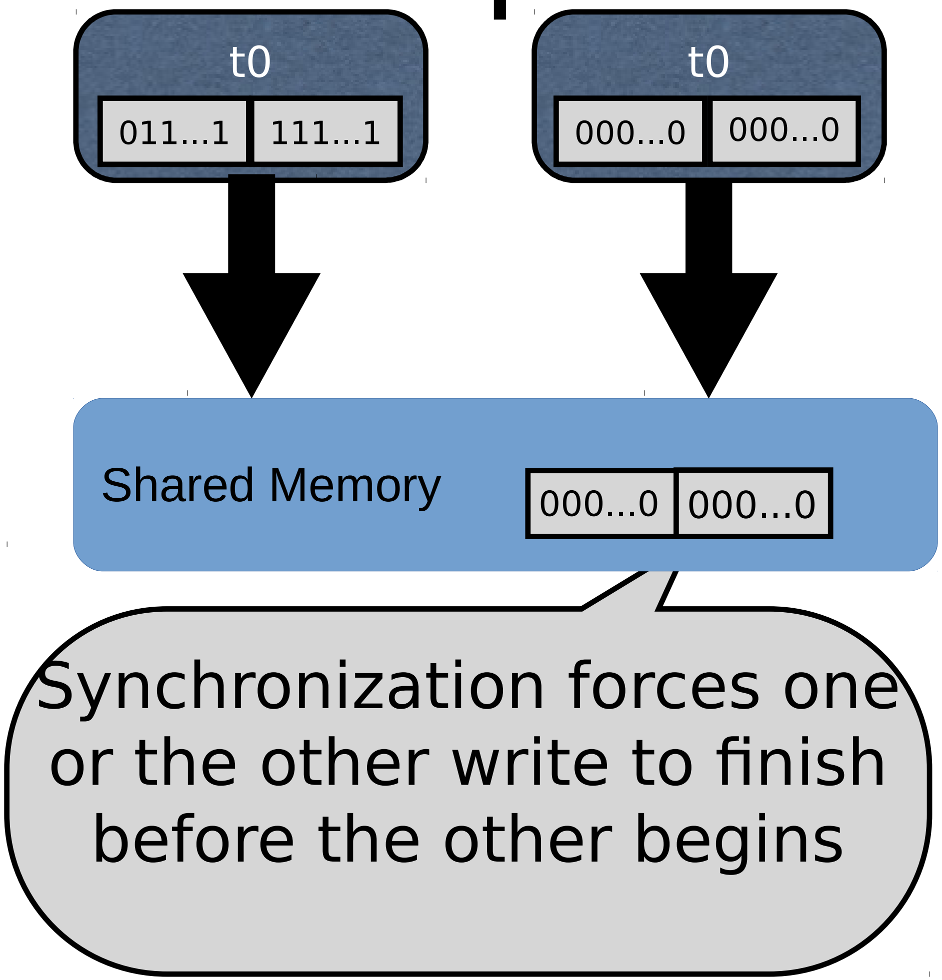
The wrong thing happens



The wrong thing happens



Orders not prevented can happen - so prevent them



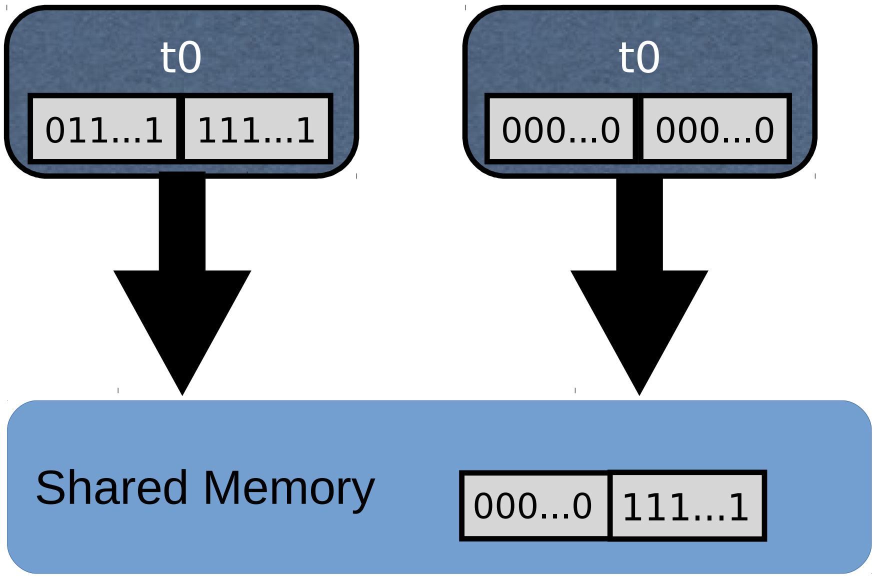
Thread 0

```
...
synchronized(C) {
    C.li = Long.MAX_VALUE( );
}
```

Thread 1

```
...
synchronized(C) {
    C.li = 0;
}
```

Not just a Java problem



This problem will occur with any language unless

1. the language spec/compiler enforce the atomicity of the writes
2. the hardware enforces atomicity of multi-word writes (and program will not be portable)

Why don’t all language specs prevent this?

- The problem has three sources:
  1. The program has a race
  2. Synchronization is not free
  3. Writing specs that cover what a racy program means is hard, as in really, really hard
- Programmers should not write racy programs unless they really, *really,* REALLY know what they are doing -- and even then they probably don’t (search for *double-lock idiom*)
- If atomicity for atomics is provided by default, all stores of multi-word primitives will be slower to help poorly written programs

From the Java Language Specification, 2nd edition, Chapter 17

In the absence of explicit synchronization, an implementation is free to update the main memory in an order that may be surprising. *Therefore the programmer who prefers to avoid surprises should use explicit synchronization.*

Join/Wait/Notify/NotifyAll

- These are all Java provided methods to allow you to control the execution order of threads.

```
Thread t1 = new Thread(. . .);
...
t1.join( )
```

- This code blocks until t1 completes. *join* is inherited from a thread class
- **`join`**`(long millis)` waits *millis* milliseconds for the thread to die
- **`join`**`(long millis, int nanos)` waits *millis* milliseconds and *nanos* nanoseconds for the thread to die

Wait( )

- A method in Object
- Puts the thread that executes the wait method in the *wait* queue associated with the object's monitor (lock) where it stays until another thread executes a notify (and it is chosen) or notifyAll or it is interrupted
  - the thread wanting to wait must own the monitor
  - threads own monitors
    - By executing a synchronized instance method of that object.
    - By executing the body of a synchronized statement that synchronizes on the object.
    - For objects of type Class, by executing a synchronized static method of that class.

Notify

- A method in Object
- notify - wake up a single thread waiting on the executing object's monitor. You don't get to pick the thread. The woken up thread acquires the monitor.
  - the notifying thread must own the monitor
  - the notified thread competes with other threads to acquire the monitor as soon as the notifying thread relinquishes it
- *notifyAll* wakes up all such threads and puts them all into the locks *blocked* queue. Only one notified thread will acquire the lock and continue on. The others will wait in the blocked queue for the lock to be released and then acquire it, one-by-one.
- Use notify when all threads accessing the resource are the same. Use notifyAll otherwise. ***As a general rule, if no specific reason to use notify, use notifyAll.***

Stop

`stop()`

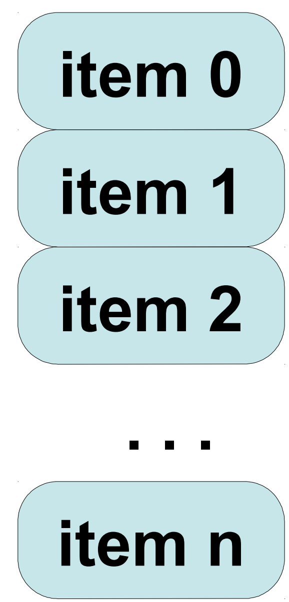
Deprecated. This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of stop should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the interrupt method should be used to interrupt the wait.

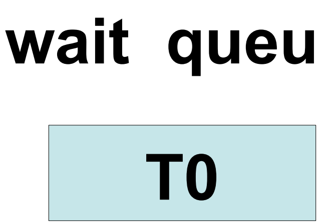
Notify/Wait example

```
public class BlockingQueue<T> {

    private Queue<T> queue = new LinkedList<T>();
    private int capacity;

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }
    // code to add and remove elements to/from the queue
}
```

Notify/Wait example continued

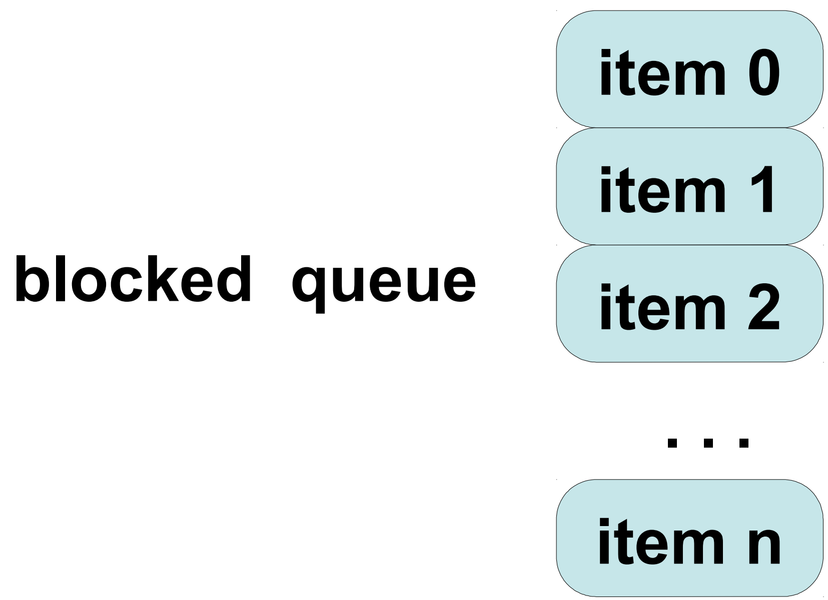
```java
public synchronized void put(T element) throws
    InterruptedException {
    while(queue.size() == capacity) {
        wait();
    }

    queue.add(element);
    notifyAll();
}
```

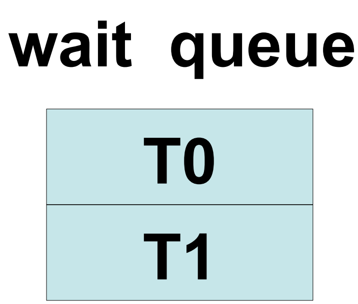




Notify/Wait example continued

```
public synchronized void put(T element) throws
    InterruptedException {
    while(queue.size() == capacity) {
        wait();
    }

    queue.add(element);
    notifyAll();
}
```





Notify/Wait example continued

```
public synchronized T take() throws
    InterruptedException {
    while(queue.isEmpty()) {
        wait();
    }

    T item = queue.remove();
    notifyAll();
    return item;
}
```