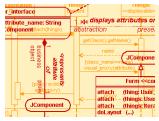
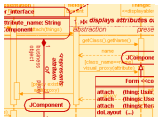
Singletons and Threading



Classic Singleton (Doesn't Work)

T1 T2

```
class Singleton           // a one-of-a-kind object
{
     static Singleton instance;
     static getInstance()
     {
          if( instance == null )
             instance = new Singleton();

          return instance;
     }
}
```

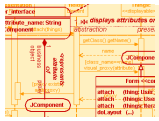



Classic Singleton (Doesn't Work)

T2

```
class Singleton           // a one-of-a-kind object
{
     static Singleton instance;
     static getInstance()
     {
          if( instance == null )
             instance = new Singleton();

          return instance;
     }
}
```

T1

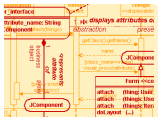


Classic Singleton (Doesn't Work)

```
class Singleton           // a one-of-a-kind object
{
     static Singleton instance;
     static getInstance()
     {
          if( instance == null )
             instance = new Singleton();

          return instance;
     }
}
```

T1

T2

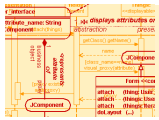


Classic Singleton (Doesn't Work)

```
class Singleton            // a one-of-a-kind object
{
    static Singleton instance;
    static getInstance()
    {
        if( instance == null )
            instance = new Singleton();

        return instance;
    }
}
```

T1

2-4



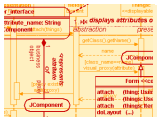
Classic Singleton (Doesn't Work)

```
class Singleton            // a one-of-a-kind object
{
    static Singleton instance;
    static getInstance()
    {
        if( instance == null )
            instance = new Singleton();

        return instance;
    }
}
```

T1

There are now two of them!

2-5



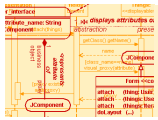
The simplest solution

T1 T2

```
static Singleton instance;
Singleton getInstance()
{
    lock( typeof(Singleton) )
    {   if( instance == null
            instance = new Singleton();
        return instance;
    }
}
```

The type object is itself a singleton: There's only one and it's globally accessible. Could also create a `static object` and lock on that.

3-1



The simplest solution

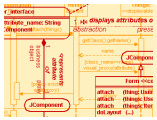
T2

T1

```
static Singleton instance;
Singleton getInstance()
{
    lock( typeof(Singleton) )
    {   if( instance == null
            instance = new Singleton();
        return instance;
    }
}
```

The type object is itself a singleton: There's only one and it's globally accessible. Could also create a `static object` and lock on that.

3-2



The simplest solution

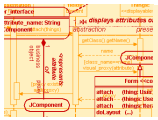
T2

T1

```
static Singleton instance;
Singleton getInstance()
{
   lock( typeof(Singleton) )
   {   if( instance == null
          instance = new Singleton();
       return instance;
   }
}
```

The type object is itself a singleton: There's only one and it's globally accessible. Could also create a `static object` and lock on that.

3-3



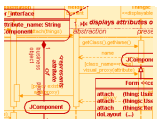
The simplest solution

T2

```
static Singleton instance;
Singleton getInstance()
{
   lock( typeof(Singleton) )
   {   if( instance == null
          instance = new Singleton();
       return instance;
   }
}
```

The type object is itself a singleton: There's only one and it's globally accessible. Could also create a `static object` and lock on that.

3-4



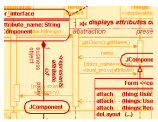
The simplest solution

T2

```
static Singleton instance;
Singleton getInstance()
{
   lock( typeof(Singleton) )
   {   if( instance == null
          instance = new Singleton();
       return instance;
   }
}
```

The type object is itself a singleton: There's only one and it's globally accessible. Could also create a `static object` and lock on that.

3-5

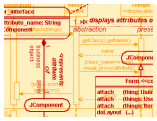


The simplest solution

```
static Singleton instance;
Singleton getInstance()
{
   lock( typeof(Singleton) )
   {   if( instance == null
          instance = new Singleton();
       return instance;
   }
}
```

The type object is itself a singleton: There's only one and it's globally accessible. Could also create a `static object` and lock on that.

3-6

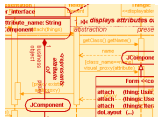


Atomicity ≠ visibility

Do not confuse atomicity with visibility.
Double-checked locking does not work!

```
static Singleton instance;
Singleton GetInstance()
{
    if( instance == null )
    {   lock( typeof(Singleton) )
        {  if( instance == null )
              instance = new Singleton();
        }
    }
    return instance;
}
```


4-1

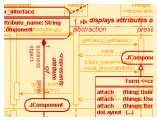


Atomicity ≠ visibility

Do not confuse atomicity with visibility.
Double-checked locking does not work!

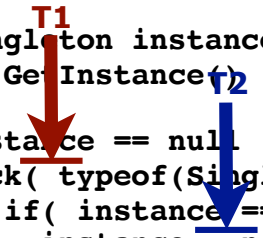

```
static Singleton instance;
Singleton GetInstance()
{
    if( instance == null )
    {   lock( typeof(Singleton) )
        {  if( instance == null )
              instance = new Singleton();
        }
    }
    return instance;
}
```


4-2



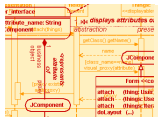
Atomicity ≠ visibility

Do not confuse atomicity with visibility.
Double-checked locking does not work!




```
static Singleton instance;
Singleton GetInstance()
{
    if( instance == null )
    {   lock( typeof(Singleton) )
        {  if( instance == null )
              instance = new Singleton();
        }
    }
    return instance;
}
```


4-3

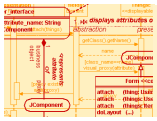


Atomicity ≠ visibility

Do not confuse atomicity with visibility.
Double-checked locking does not work!


```
static Singleton instance;
Singleton GetInstance()
{
    if( instance == null )
    {   lock( typeof(Singleton) )
        {  if( instance == null )
              instance = new Singleton();
        }
    }
    return instance;
}
```


4-4

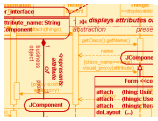


Atomicity ≠ visibility

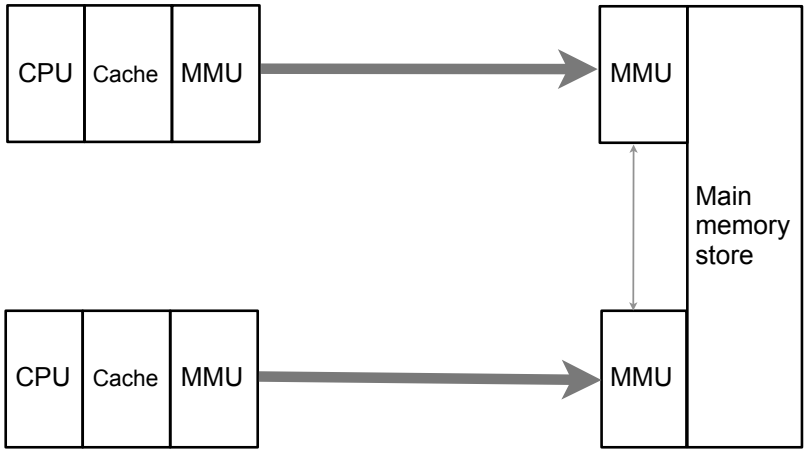
Do not confuse atomicity with visibility.
Double-checked locking does not work!

```
static Singleton instance;
Singleton GetInstance()
{
    if( instance == null )
    {   lock( typeof(Singleton) )
        {  if( instance == null )
               instance = new Singleton();
        }
    }
    return instance;
}
```

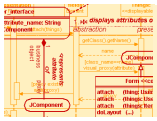

4-5



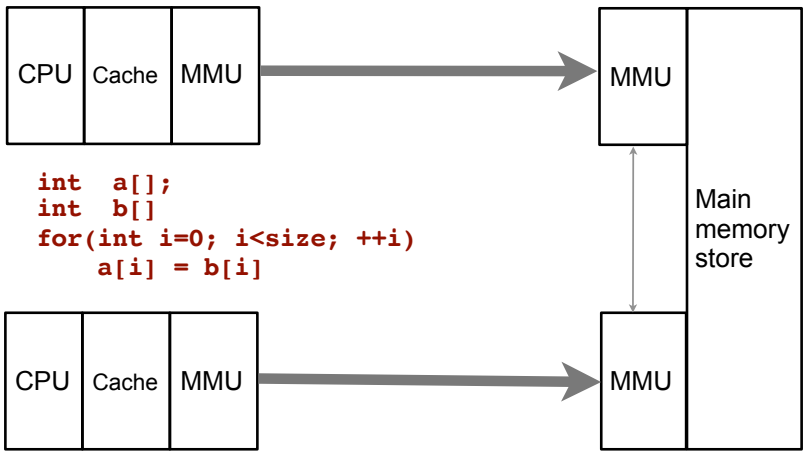
The problem's in the hardware



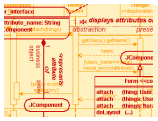
5-1



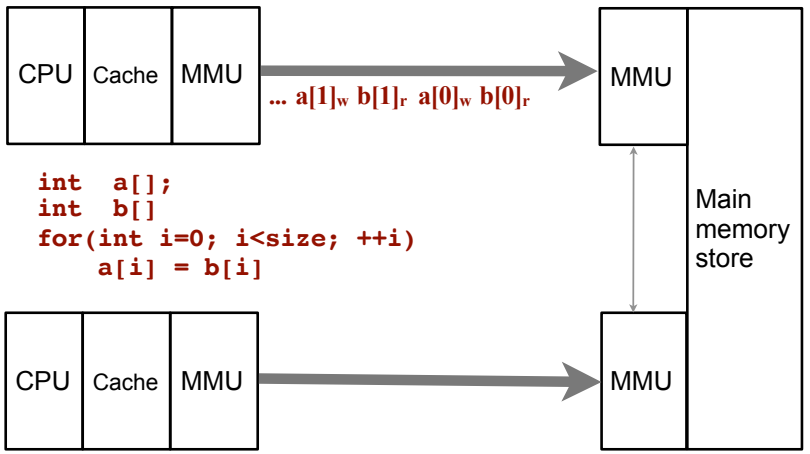
The problem's in the hardware



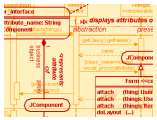
5-2



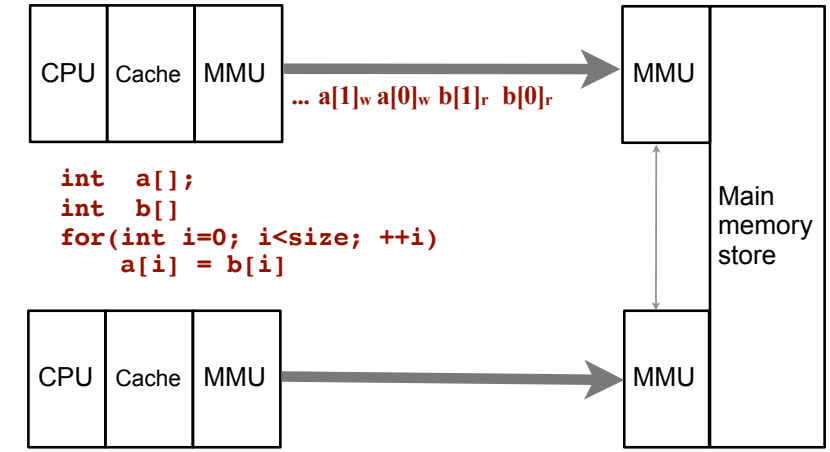
The problem's in the hardware



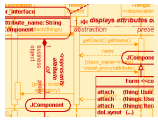
5-3



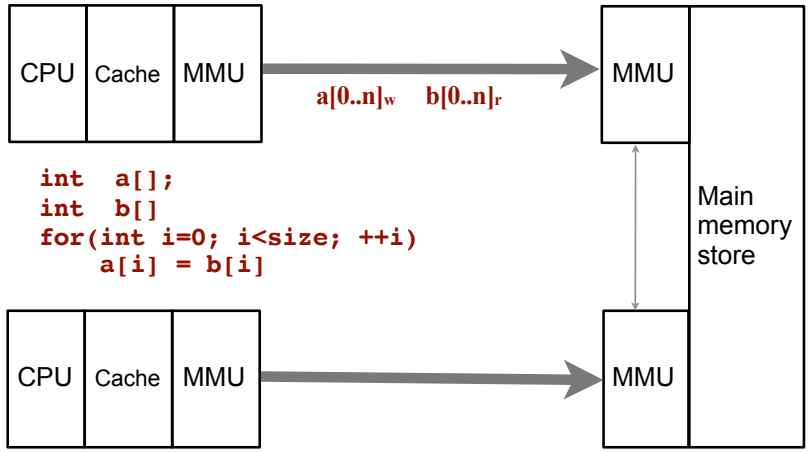
The problem's in the hardware



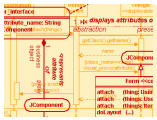
5-4



The problem's in the hardware



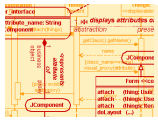
5-5



Double-checked locking revisited

```
static Singleton instance;
Singleton GetInstance()
{
    if( instance == null )
    {   lock( typeof(Singleton) )
        {  if( instance == null )
              instance = new Singleton();
        }
    }
    return instance;
}
```


6-1

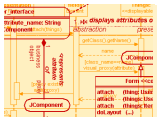


Double-checked locking revisited

```
static Singleton instance;
Singleton GetInstance()
{
    if( instance == null )
    {   lock( typeof(Singleton) )
        {  if( instance == null )
              instance = new Singleton();
        }
    }
    return instance;
}
```

The assignments made in this constructor...

6-2



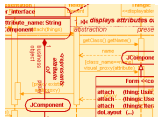
Double-checked locking revisited

```
static Singleton instance;
Singleton GetInstance()
{
    if( instance == null )
    {   lock( typeof(Singleton) )
        {  if( instance == null )
              instance = new Singleton();
        }
    }
    return instance;
}
```

The assignments made in this constructor...

...may happen before or after this assignment

6-3



Double-checked locking revisited

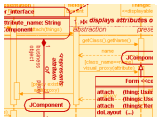
```
static Singleton instance;
Singleton GetInstance()
{
    if( instance == null )
    {   lock( typeof(Singleton) )
        {  if( instance == null )
              instance = new Singleton();
        }
    }
    return instance;
}
```

The assignments made in this constructor...

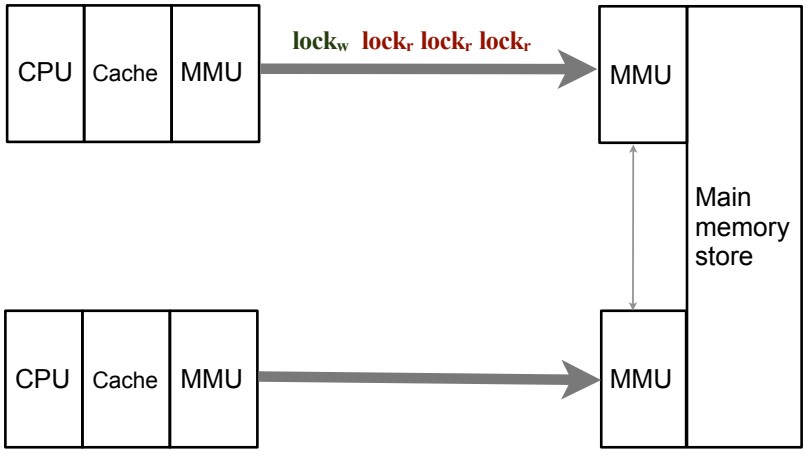
...may happen before or after this assignment

Even if single threaded: The optimizer may move the assignment.

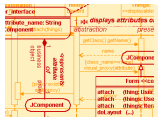
6-4



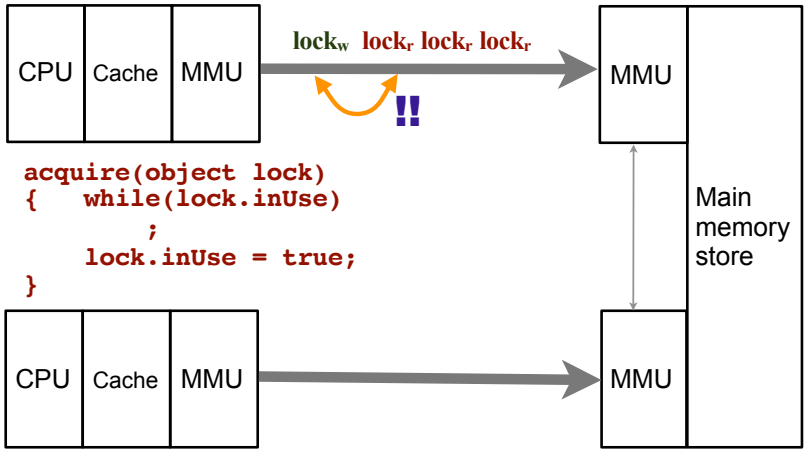
memory barriers



7-1

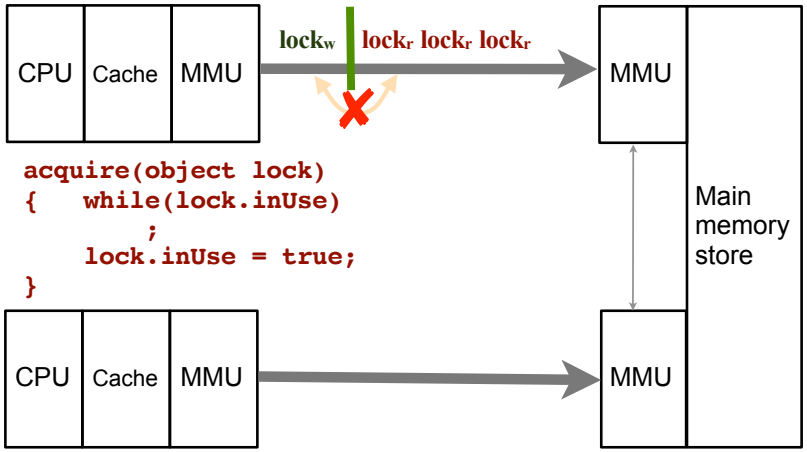


memory barriers



7-2

memory barriers



Still, don't use flags

```
boolean amWriting = false;

amWriting = true;
// bunch of assignments
amWriting = false;
```

these assignments are all subject to reordering.

Manual memory barriers

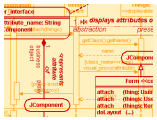
```
public static void VolatileWrite
                    (ref int address, int value)
{
    MemoryBarrier();    //establishes full barrier!
    address = value;
}
public static int VolatileRead(ref int address)
{
    int num = address;
    MemoryBarrier();
    return num;
}
```

These methods are in the `Thread` class.

Not significantly different from `lock()`.

Fix it (ver. 1)

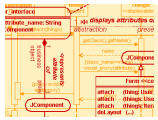
```
static Singleton instance;
Singleton getInstance()
{
    lock( typeof(Singleton) )
    {   if( instance == null
            instance = new Singleton();
        return instance;
    }
}
```



Fix it (ver. 1)

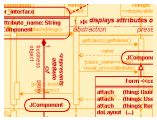
```
static Singleton instance;
Singleton getInstance()
{
    lock( typeof(Singleton) )
    {   if( instance == null
            instance = new Singleton();
        return instance;
    }
}
```

To guarantee visibility: the reading thread should acquire a lock that was held by the writing thread.



Solution 2 (Java or C#)

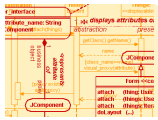
```
private sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();
    public static Instance getInstance()
    {   return instance;
    }

    private Singleton(){}
}
```

Solution 2 (Java or C#)

```
private sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();
    public static Instance getInstance()
    {   return instance;
    }

    public static readonly int X = 10;
    private Singleton(){}
}
```

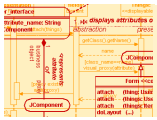
Accessing X loads instance as a side effect!



Solution 2 (Java or C#)

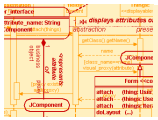
```
private sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();
    public static Instance getInstance()
    {   return instance;
    }

    private class Nested
    {  static Nested(){}
       internal static readonly Singleton instance = new Singleton();
    }
    public static readonly int X = 10;
    private Singleton(){}
}
```

Static constructor tells compiler not to mark type as `beforefieldinit`



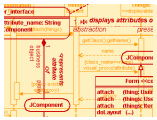
Solution 2 (Java or C#)

```
private sealed class Singleton
{
    public static Instance getInstance()
    {   return Nested.instance;
    }
   private class Nested
   {  static Nested(){}
      internal static readonly Singleton instance = new Singleton();
   }
   public static readonly int X = 10;
   private Singleton(){}
}
```

Solution 3 (C# only)

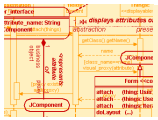
```
public sealed class Singleton
{
  private static readonly
  Lazy<Singleton> lazy = new Lazy<Singleton>
                         (() => new Singleton());

  public static Singleton getInstance()
  {  return lazy.Value;
  }

  private Singleton(){}
}
```

volatile

```
volatile Singleton instance;
```

read followed by write	read always fetches old value
read followed by read	both reads yield same value
write followed by write	value after first write not visible after second write
write followed by read	**read might see the old value (the one before the write)**



Still, don't use flags

```
volatile boolean amWriting

amWriting = true;
// bunch of assignments
amWriting = false;
```

these assignments are still subject to reordering.