



aicas Technology

Multicore Systems: Impact of the Programming Language



Dr. Fridtjof Siebert, CTO
aicas GmbH
ESA ESTEC ADCSS, 27th October 2011



Typical Problems on a Multicore

- typical code sequence (C/C++ or Java)

```
int counter;
```

```
void increment ()  
{  
    counter++;  
}
```



Typical Problems on a Multicore

- typical code sequence (C/C++ or Java)

```
int counter;
```

```
void increment ()  
{  
    counter++;  
}
```

{

```
    r1 = counter;  
    r2 = r1 + 1;  
    counter = r2;  
}
```



Typical Problems on a Multicore

- typical code sequence (C/C++ or Java)

```
int counter;
```

```
void increment()  
{  
    counter++;  
}
```

	Thread 1	Thread 2
{	r1 = counter;	r1 = counter;
	r2 = r1 + 1;	r2 = r1 + 1;
	counter = r2;	counter = r2;



Typical Problems on a Multicore

- typical code sequence (C/C++ or Java)

```
int counter;
```

```
void increment()  
{  
    counter++;  
}
```

Thread 1

```
r1 = counter;  
r2 = r1 + 1;  
counter = r2;
```

Thread 2

```
r1 = counter;  
r2 = r1 + 1;  
counter = r2;
```

One increment() can get lost!



Typical Problems on a Multicore

- typical code sequence (C/C++ or Java)

```
int counter;
```

```
void increment ()  
{  
    counter++;  
}
```





Typical Problems on a Multicore

- typical code sequence (C/C++ or Java)

```
int counter;  
  
void increment ()  
{  
    counter++;  
}
```



- this code misses synchronization
- but on a single core, it practically always works!
- on a multicore, chances for failure explode!



Synchronization

- solution: synchronize

```
int counter;
```

```
synchronized void increment ()  
{  
    counter++;  
}
```

- easy, problem solved.
- Or? See later.



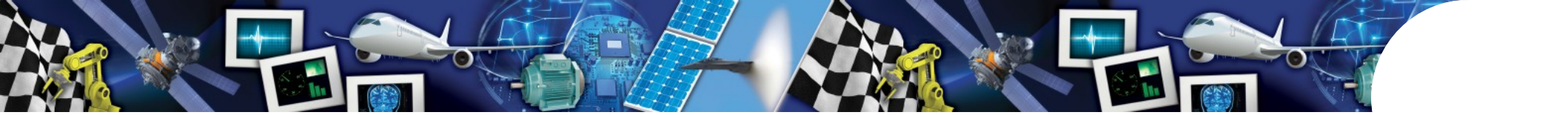
Atomic Operations

- What is the result of

```
int a, b;  /* 32 bit, initially 0 */
```

<u>Thread 1</u>	<u>Thread 2</u>
<code>b = a;</code>	<code>a = -1;</code>

?



Atomic Operations

- What is the result of

```
int a, b;  /* 32 bit, initially 0 */
```

<u>Thread 1</u>	<u>Thread 2</u>
<code>b = a;</code>	<code>a = -1;</code>

?

- `b == 0`
`b == -1`



Atomic Operations

- What is the result of

```
long a, b; /* 64 bit, initially 0 */
```

<u>Thread 1</u>	<u>Thread 2</u>
b = a;	a = -1;

?



Atomic Operations

- What is the result of

```
long a, b; /* 64 bit, initially 0 */
```

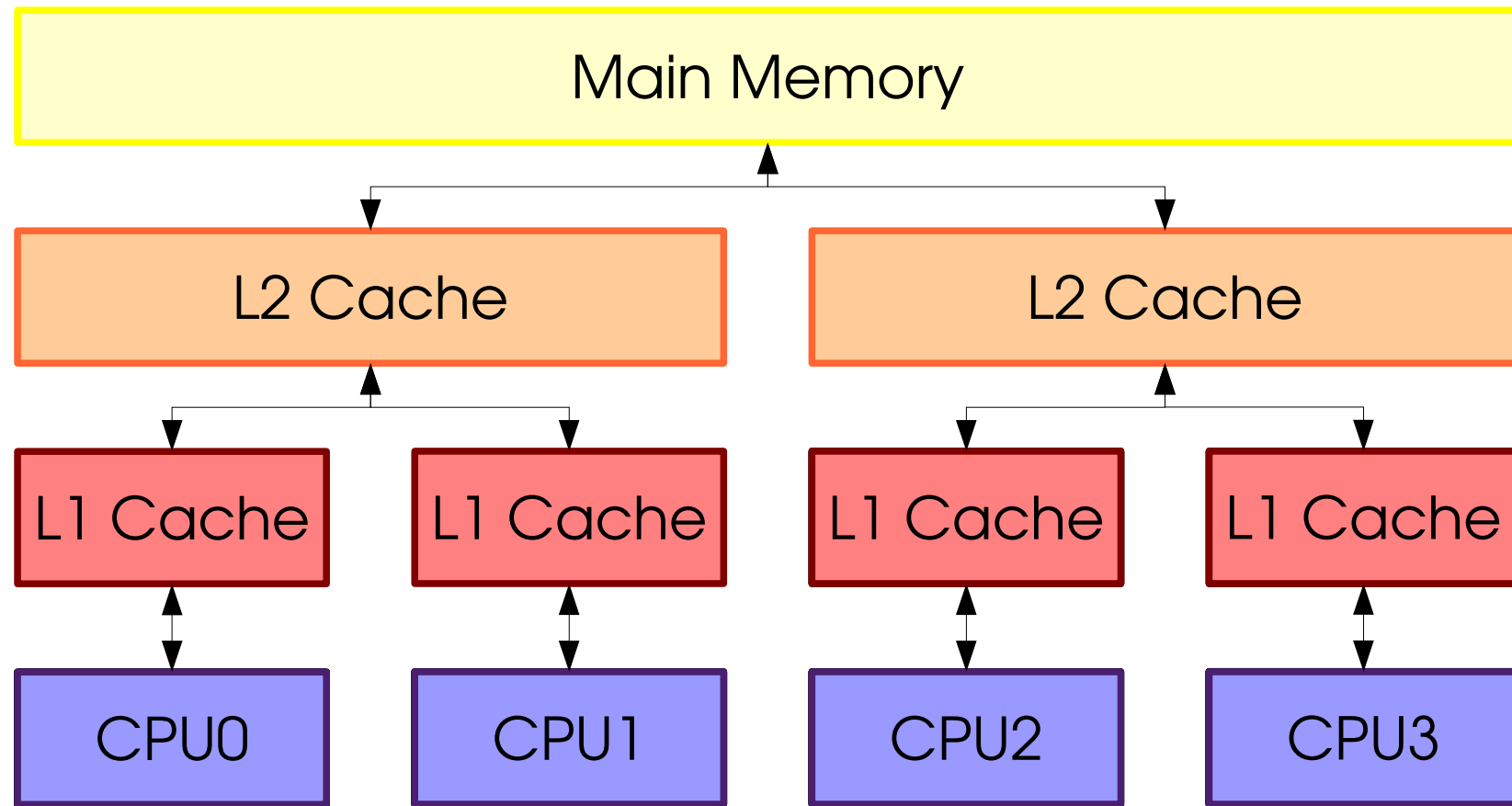
<u>Thread 1</u>	<u>Thread 2</u>
<code>b = a;</code>	<code>a = -1;</code>

?

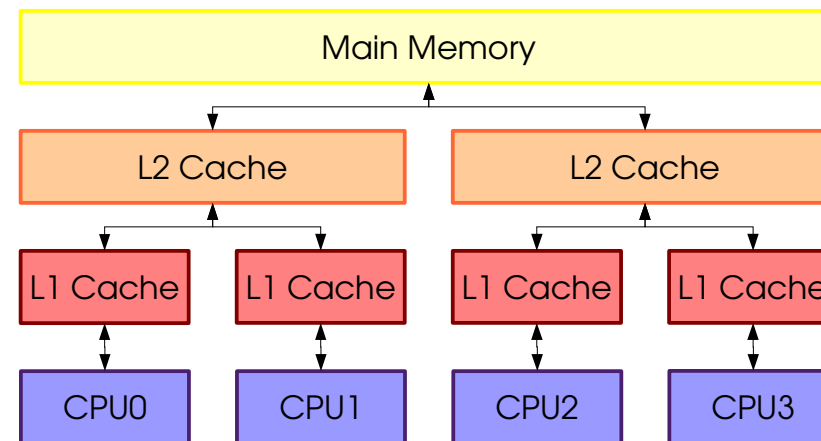
- `b == 0`
- `b == -1`
- `b == -4294967296`
- `b == 4294967295`

Cache Structure

- CPUs use local caches for performance



Cache Structure



- Modifications do not become visible immediately
- Modifications may be re-ordered
- Reads may refer to outdated (cached) data
- Reads may be re-ordered



Typical Problems on a Multicore

- polling update

```
long counter;  
[...]  
do  
{  
    doSomething();  
}  
while (counter < MAX);
```


- **counter** is incremented by parallel thread
- on a Multicore, changes to counter may not become visible!



Typical Problems on a Multicore

- polling update

```
long counter;  
[...]  
do  
{  
    do_something();  
}  
while (counter < MAX);
```



- **counter** is incremented by parallel thread
- on a Multicore, changes to counter may not become visible!



Solution: volatile?

- polling update

```
volatile long counter;  
[...]  
do  
{  
    doSomething();  
}  
while (counter < MAX);
```

- works for Java





Solution: volatile?

- polling update

```
volatile long counter;  
[...]  
do  
{  
    doSomething();  
}  
while (counter < MAX);
```

- works for Java
- does not work for C!





We must understand the memory model!

- Memory model specifies what optimisations are permitted by the compiler or underlying hardware
- C/C++ programs have undefined semantics in case of race conditions
- Java defines a strict memory model



Java's memory model

- ordering operations are
 - ♦ synchronized block
 - ♦ accessing a volatile variable
- The presence of an ordering operation determines the visible state in shared memory



Java's memory model: Enforcing Order

- all reads are completed before
 - ♦ entering synchronized block, or
 - ♦ reading a volatile variable

|||||➡ **read fence**

- all writes are completed before
 - ♦ exiting a synchronized block, or
 - ♦ writing a volatile var

|||||➡ **write fence**



Java's memory model: Data Races

- data races are not forbidden in Java
 - ♦ you can use shared memory variables
 - ♦ your code has to tolerate optimizations
- examples
 - ♦ collecting debugging / profiling information
 - ♦ useful if occasional errors due to data races are tolerable



Example use of Java's memory model

- Shared memory communication

```
Ptr      p;  
boolean  p_valid;
```

Thread 1

```
p = new Ptr();  
p_valid = true;
```



Example use of Java's memory model

- Shared memory communication

```
Ptr      p;  
boolean  p_valid;
```

Thread 1

Thread 2

```
p = new Ptr();  
p_valid = true;
```



Example use of Java's memory model

- Shared memory communication

```
Ptr    p;  
boolean p_valid;
```

Thread 1

```
p = new Ptr();  
p_valid = true;
```

Thread 2

```
if (p_valid)  
    p.call();
```



Example use of Java's memory model

- Shared memory communication

Thread 1

```
p = new Ptr();  
p_valid = true;
```

Thread 2

```
if (p_valid)  
    p.call();
```




Example use of Java's memory model

- Shared memory communication

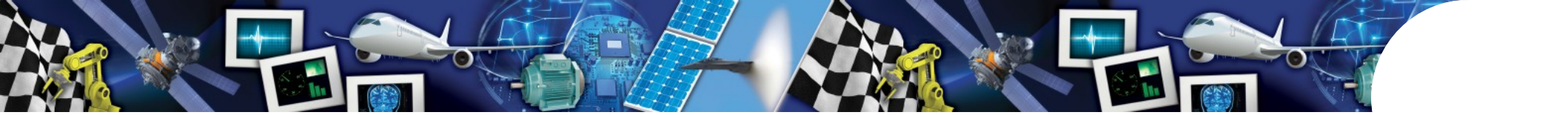
Thread 1

```
p = new Ptr();  
p_valid = true;
```

Thread 2

```
if (p_valid)  
    p.call();
```

What may happen:



Example use of Java's memory model

- Shared memory communication

Thread 1

```
p = new Ptr();  
p_valid = true;
```

Thread 2

```
if (p_valid)  
    p.call();
```

What may happen:

```
t1 = new Ptr();  
t2 = true;  
p_valid = t2;  
p = t1;
```



Example use of Java's memory model

- Shared memory communication

Thread 1

```
p = new Ptr();  
p_valid = true;
```

Thread 2

```
if (p_valid)  
    p.call();
```

What may happen:

```
t1 = new Ptr();  
t2 = true;  
p_valid = t2;  
p = t1;
```

Writes reordered!



Example use of Java's memory model

- Shared memory communication

Thread 1

~~`p = new Ptr();
p_valid = true;`~~

Thread 2

```
if (p_valid)  
    p.call();
```

What may happen:

```
t1 = new Ptr();  
t2 = true;  
p_valid = t2;  
p = t1;
```

Writes reordered!



Example use of Java's memory model

- Shared memory communication

Thread 1

~~`p = new Ptr();
p_valid = true;`~~

Thread 2

`if (p_valid)
p.call();`

What may happen:

`t1 = new Ptr();
t2 = true;
p_valid = t2;
p = t1;`

Writes reordered!

`t3 = p;
if (p_valid)
t3.call();`

Reads reordered!



Example use of Java's memory model

- Shared memory communication

Thread 1

~~`p = new Ptr();
p_valid = true;`~~

What may happen:

```
t1 = new Ptr();  
t2 = true;  
p_valid = t2;  
p = t1;
```

Writes reordered!

Thread 2

~~`if (p_valid)
p.call();`~~

```
t3 = p;  
if (p_valid)  
    t3.call();
```

Reads reordered!



Example use of Java's memory model

- Shared memory communication



```
volatile Ptr    p;  
volatile boolean p_valid;
```

Thread 1

```
p = new Ptr();  
p_valid = true;
```

Thread 2

```
if (p_valid)  
    p.call();
```

in Java



Example use of Java's memory model

- Shared memory communication

```
volatile Ptr      p;  
volatile boolean p_valid;
```

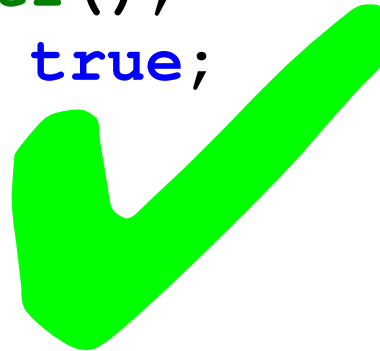
Thread 1

```
p = new Ptr();  
p_valid = true;
```

in Java

Thread 2

```
if (p_valid)  
    p.call();
```





Example use of Java's memory model

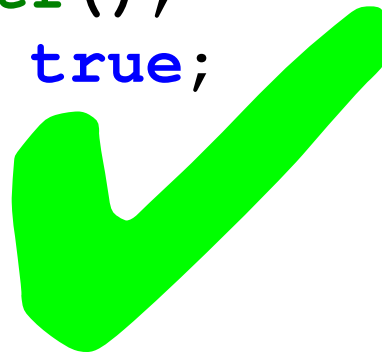
- Shared memory communication

```
volatile Ptr      p;  
volatile boolean p_valid;
```

Thread 1

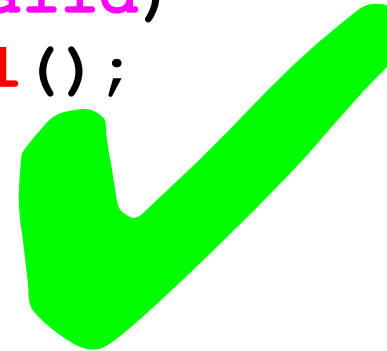
```
p = new Ptr();  
p_valid = true;
```

in Java



Thread 2

```
if (p_valid)  
    p.call();
```





Example use of C's memory model

- Shared memory communication

```
volatile Obj      *p;  
volatile boolean p_valid;
```

Thread 1

```
p = malloc(..);  
p_valid = TRUE;
```

Thread 2

```
if (p_valid)  
    p->f = ..;
```

in C?



Example use of C's memory model

- Shared memory communication

```
volatile Obj      *p;  
volatile boolean p_valid;
```

Thread 1

```
p = malloc(..);  
p_valid = TRUE;
```

Thread 2

```
if (p_valid)  
    p->f = ..;
```

in C?

CPU may still reorder memory accesses!



Example use of C's memory model

- Shared memory communication

```
volatile Obj      *p;  
volatile boolean p_valid;
```

Thread 1



```
p = malloc(sizeof Obj);  
p_valid = TRUE;
```

Thread 2

```
if (p_valid)  
    p->f = ..;
```

in C?

CPU may still reorder memory accesses!



Example use of C's memory model

- Shared memory communication

```
volatile Obj      *p;  
volatile boolean p_valid;
```

Thread 1

~~```
p = malloc(...);
p_valid = TRUE;
```~~

in C?

Thread 2

~~```
if (p_valid)  
p = ...;
```~~

CPU may still reorder memory accesses!



Example use of C's memory model

- Shared memory communication

```
volatile Obj      *p;  
volatile boolean p_valid;
```

Thread 1

```
p = malloc(...);  
p_valid = TRUE;
```

Thread 2

```
if (p_valid)  
    p->f = ...;
```

How to fix it? Add memory fences!



Example use of C's memory model

- Shared memory communication

```
volatile Obj      *p;  
volatile boolean p_valid;
```

Thread 1

```
p = malloc(..);  
asm volatile(  
    "sfence" ::: "memory");  
p_valid = TRUE;
```

Thread 2

```
if (p_valid)  
    p->f = ..;
```

How to fix it? Add memory fences!



Example use of C's memory model

- Shared memory communication

```
volatile Obj      *p;  
volatile boolean p_valid;
```

Thread 1

```
p = malloc(..);  
asm volatile(  
    "sfence" ::: "memory");  
p_valid = TRUE;
```

Thread 2

```
if (p_valid)  
{  
    asm volatile(  
        "lfence" ::: "memory");  
    p->f = ..;  
}
```

How to fix it? Add memory fences!



Example use of C's memory model

- Shared memory communication

```
volatile Obj      *p;  
volatile boolean p_valid;
```

Thread 1

```
p = malloc(...);  
asm volatile(  
    "sfence" ::: "memory");  
p_valid = TRUE;
```

Thread 2

```
if (p_valid)  
{  
    asm volatile(  
        "lfence" ::: "memory");  
    p->f = ..;  
}
```

How to fix it? Add memory fences!



Out-of-thin-Air

- imagine this code

```
int x = 0, n = 0;
```

Thread 1

```
for (i=0; i<n; i++)  
    x += f(i);
```

Thread 2

```
x = 42;  
print(x);
```




Out-of-thin-Air

- imagine this code

```
int x = 0, n = 0;
```

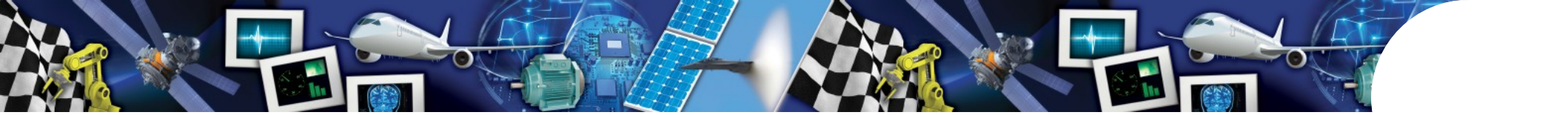
Thread 1

```
for (i=0; i<n; i++)  
    x += f(i);
```

Thread 2

```
x = 42;  
print(x);
```

- can only print 42 in Java



Out-of-thin-Air: Introduction of Writes

- loop optimization in C/C++

```
int x = 0, n = 0;
```

Thread 1

```
tmp = x;  
for (i=0; i<n; i++)  
    tmp += f(i);  
x = tmp;
```

Thread 2

```
x = 42;  
  
print(x);
```



Out-of-thin-Air: Introduction of Writes

- loop optimization in C/C++

```
int x = 0, n = 0;
```

Thread 1

```
tmp = x;  
for (i=0; i<n; i++)  
    tmp += f(i);  
x = tmp;
```

Thread 2

```
x = 42;  
  
print(x);
```

- can print 0 in C/C++



Out-of-thin-Air

- imagine this code

```
int x = 0, y = 0;
```

Thread 1

```
r1 = x;  
y = r1;
```

Thread 2

```
r2 = y;  
x = r2;
```



Out-of-thin-Air

- imagine this code

```
int x = 0, y = 0;
```

Thread 1

```
r1 = x;  
y = r1;
```

Thread 2

```
r2 = y;  
x = r2;
```

- Expected result

```
x == 0; y == 0;
```

- Only possible result in Java



Out-of-thin-Air: Optimization in C/C++

- imagine this code

```
int x = 0, y = 0;
```

Thread 1

```
y = 42;  
r1 = x;  
if (r1 != 42)  
    y = r1;
```

Thread 2

```
r2 = y;  
x = r2;
```

- Possible in new C++ MM. Results in

```
x == 42; y == 42;
```




Performance on a Multicore

- example: single-core app, 3 threads
- all threads synchronize frequently on the same lock



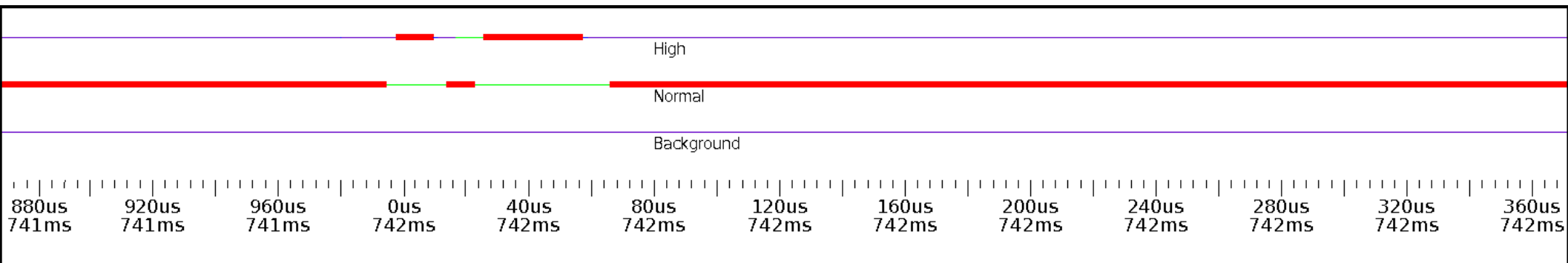
Performance on a Multicore

- example: single-core app, 3 threads
- all threads synchronize frequently on the same lock

```
while (true)
{
    synchronized (lock)
    {
        counter++;
    }
    doSomething();
}
```

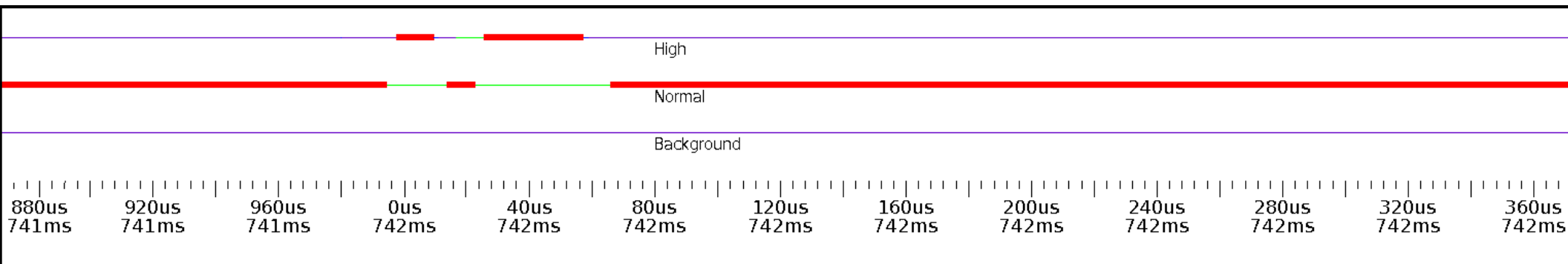
Performance on a Multicore

- example: single-core app, 3 threads

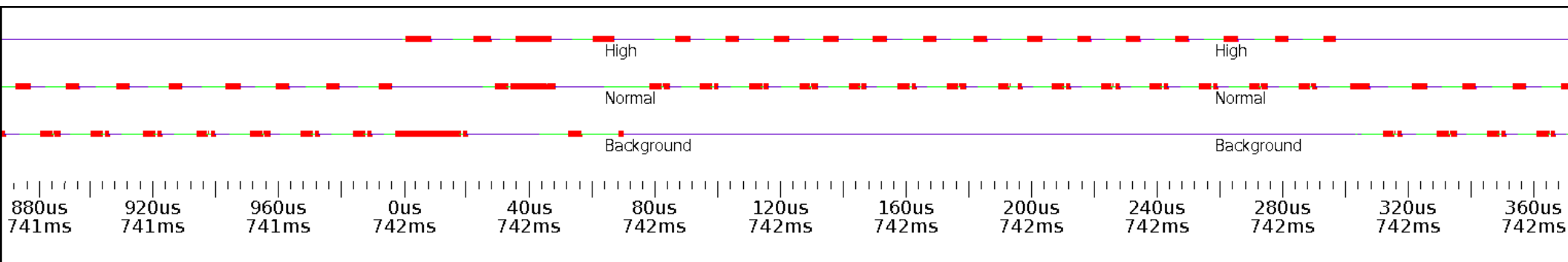


Performance on a Multicore

- example: single-core app, 3 threads



- on a multicore





Performance on a Multicore

- frequent synchronization can kill the performance
- typical non-RTOS will use heuristics to improve average performance
 - ◆ spin-lock for a short time
 - ◆ block after that



Performance on a Multicore

- can we avoid monitors?
- can we use lock-free algorithms?



Lock-free Algorithms

- typical code sequence

```
do
{
    x = counter;
    result = CAS(counter, x, x+1);
}
while (result != x);
```



Compare-And-Swap Issues

- typical code sequence

```
do
{
    x = counter;
    result = CAS(counter, x, x+1);
}
while (result != x);
```

- what is the WCET? ∞ ?



Lock-free library code

- use of libraries helps

```
AtomicInteger counter = new AtomicInteger();  
void increment()  
{  
    (void) counter.incrementAndGet();  
}
```

- Code is easier and safer
- Hand-made lock-free algorithms are not for every-day development



Conclusion

- Code that runs well on single CPU may fail on a multicore
- Clear semantics of concurrent code is required for safe applications
- Performance of locks may be prohibitive
- Lock-free code is very hard to get right
- A reliable memory model and good concurrent libraries are basis for multicore development.



Questions?

-
-
-