5. Concurrency

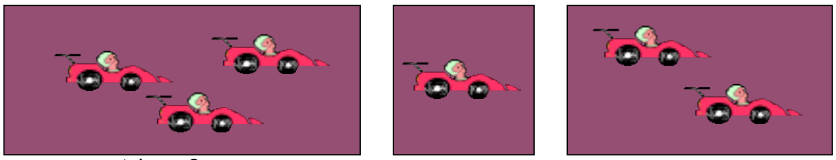
2

Programming Model

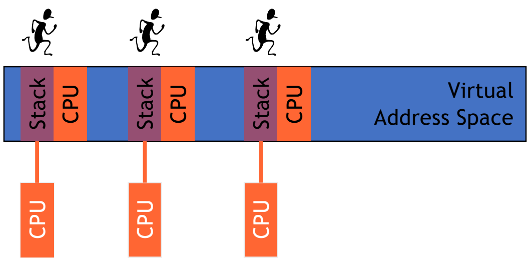
- „Unlimited” virtual memory
- „Unlimited” number of virtual CPUs
- Thread synchronization support
- Communication
  - IPC
  - Networking
- Persistent storage of information

Vision

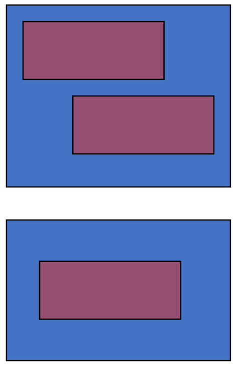
- Assume an unlimited number of available processors
- Within an application
  - Processors needed depends on the inherent parallelism / reactivity
- Between applications



CPU Multiplexing

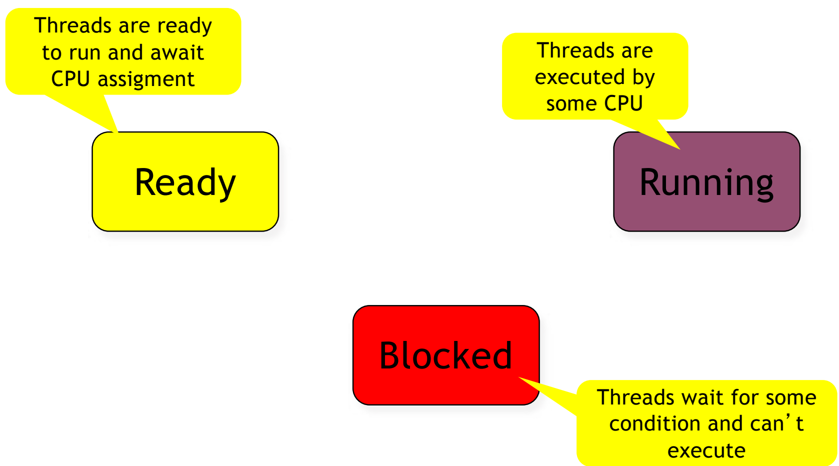


… But 1 Physical CPU Available?

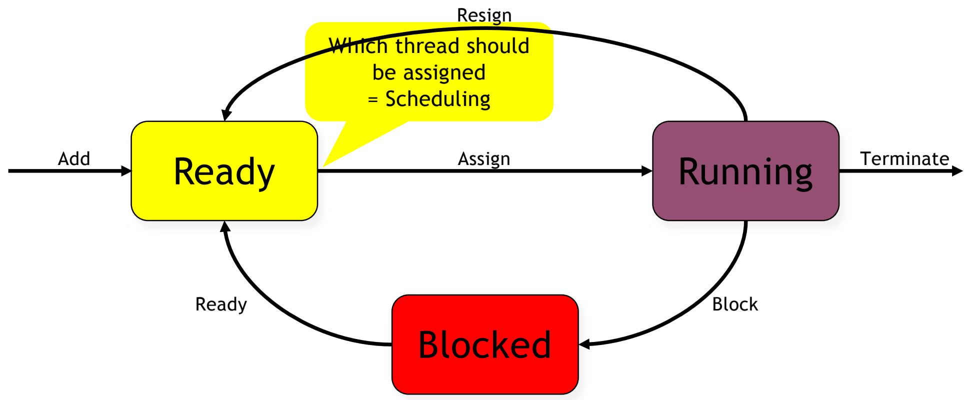


- A "Thread of Control" consists of
  - Address space
  - State of registers
  - Stack (already part of address space)
- Context Switch
- When to switch
  - Calls that might block for a longer period of time
  - Implicitly, e.g. in case of page fault or memory congestion

Model of Thread States

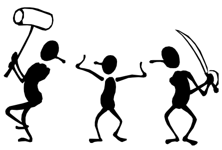


State Transistions



Competition and Cooperation

- Competition
- Applications compete for resources
  - Goal: Sole user of system
- Implicit Synchronization
  - Virtual resources
  - Serialization by OS
- Mutual exclusion



- Cooperation
- Applications (Threads) cooperate
- Sharing of resources and tasks
- Goals
  - Performance improvement
  - Avoidance of bottlenecks
  - Fault tolerance
- Mutual exclusion and more specific synchronization patterns

Autoverkehr

- 61.5 Millionen zugelassene Autos (Anfang 2014)

Polizeilich erfasste Unfälle

Straßenverkehrsunfälle

Schadensart/Ortslage	Einheit	2011	2012	2013
insgesamt	Anzahl	2 361 457	2 401 843	2 414 011
davon				
mit nur Sachschaden	Anzahl	2 055 191	2 102 206	2 122 906
mit Personenschaden	Anzahl	306 266	299 637	291 105
davon				
innerorts	Anzahl	210 427	206 696	199 650
außerorts ohne Autobahn	Anzahl	77 549	75 094	73 003
auf Autobahnen	Anzahl	18 290	17 847	18 452

Quelle: Statistisches Bundesamt


Deutsche Post
Die Post für Deutschland

Sperrgranulat

FEINSTAUB-BELASTUNG

Jedes zweite Auto in Paris steht still

Der Smog im Großraum Paris hält an und die Behörden ergreifen erstmals seit 20 Jahren drastische Maßnahmen: Nur die Hälfte der Autos und Motorräder darf auf die Straße.

17. März 2014 08:49 Uhr — 42 Kommentare

Warnhinweis an Autofahrer in Paris: Wegen des Smogs sollen sie nicht mehr als 60 km/h schnell fahren | © Francois Guillot/AFP/Getty Images

Im Kampf gegen den starken Smog beschränkt Paris erstmals seit 1997 den Autoverkehr. An diesem Montag darf nur noch die Hälfte der Fahrzeuge auf den Straßen der französischen Hauptstadt unterwegs sein. Nach Angaben von Umweltminister Philippe Martin gilt dies momentan für Autos und Motorräder, deren Kennzeichen mit einer ungeraden Zahl enden. Ausgenommen von der strengen Regelung sind Elektro- oder Hybridautos und Fahrzeuge mit mindestens drei Insassen.

Die Zeit


Bike Path
LEFT TURN
FROM
LEFT TURN
BAY ONLY


POLICE

Lock-free

Anwendungsprogrammierung

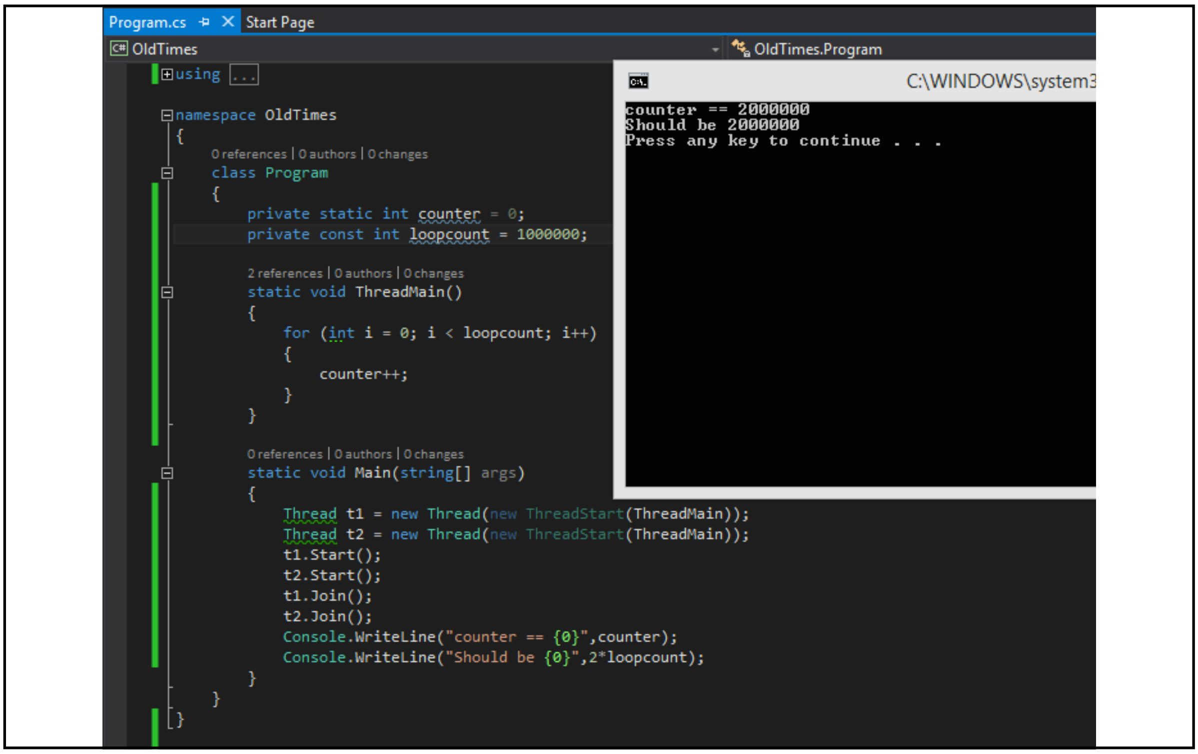
- Parallele CPUs
  - Parallele Pipelines
  - SIMD-Extensions (Single Instruction Multiple Data)
- ManyCores
  - CPUs
  - GPUs (über 10000 Kerne)
- Cluster
- Verteilte Systeme

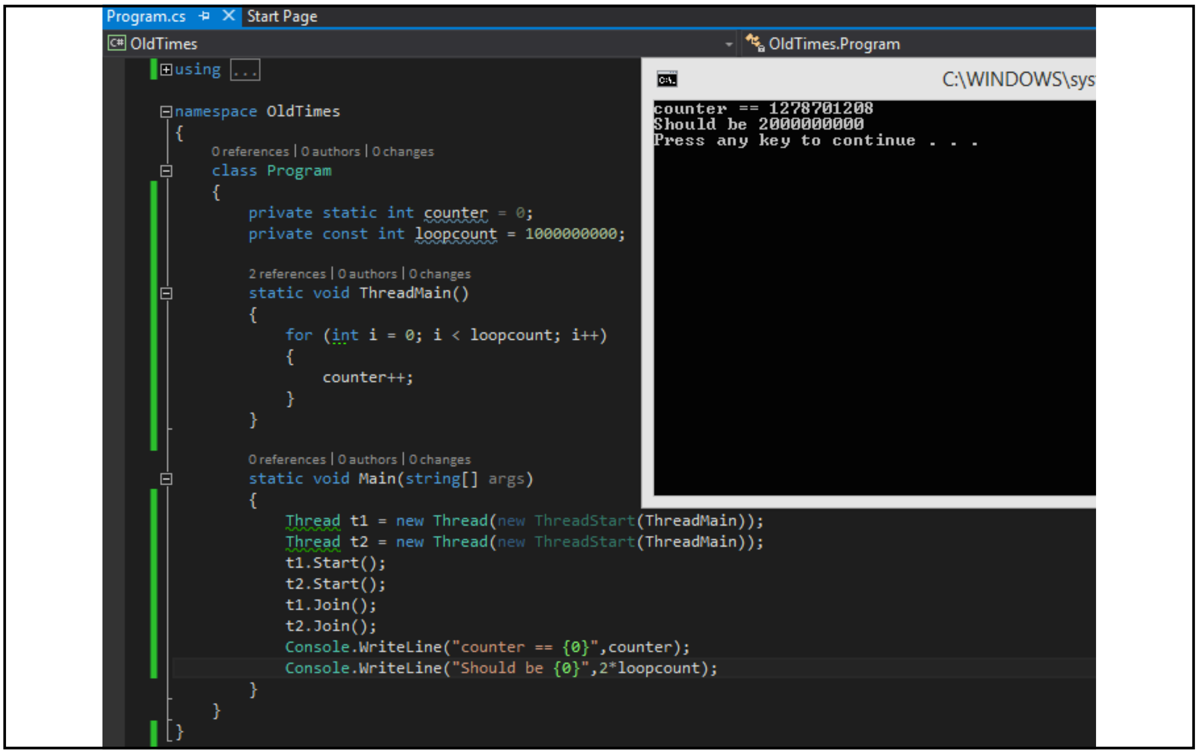
Gute alte Zeit

Threads, Mutex und Semaphore

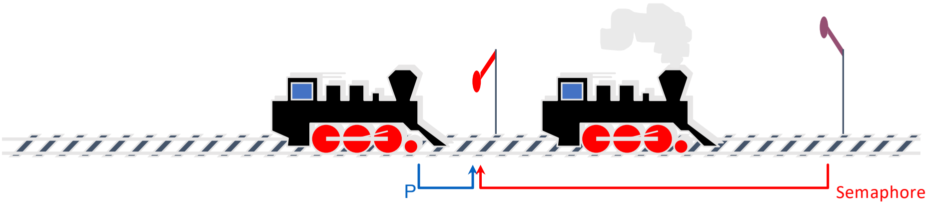
Threads

```
class Program
    {
        private static int counter = 0;
        private const int loopcount = 1000000;

        static void ThreadMain()
        {
            for (int i = 0; i < loopcount; i++)
            {
                counter++;
            }
        }

        static void Main(string[] args)
        {
            Thread t1 = new Thread(new ThreadStart(ThreadMain));
            Thread t2 = new Thread(new ThreadStart(ThreadMain));
            t1.Start();
            t2.Start();
            t1.Join();
            t2.Join();
            Console.WriteLine("counter == {0}",counter);
            Console.WriteLine("Should be {0}",2*loopcount);
        }
    }
```


Program.cs
Start Page
OldTimes
OldTimes.Program
C:\WINDOWS\system3
counter == 2000000
Should be 2000000
Press any key to continue . . .
using ...
namespace OldTimes
{
0 references | 0 authors | 0 changes
class Program
{
private static int counter = 0;
private const int loopcount = 1000000;
2 references | 0 authors | 0 changes
static void ThreadMain()
{
for (int i = 0; i < loopcount; i++)
{
counter++;
}
}
0 references | 0 authors | 0 changes
static void Main(string[] args)
{
Thread t1 = new Thread(new ThreadStart(ThreadMain));
Thread t2 = new Thread(new ThreadStart(ThreadMain));
t1.Start();
t2.Start();
t1.Join();
t2.Join();
Console.WriteLine("counter == {0}",counter);
Console.WriteLine("Should be {0}",2*loopcount);
}
}
}


Program.cs
Start Page
OldTimes
OldTimes.Program
C:\WINDOWS\sys
counter == 1278701208
Should be 2000000000
Press any key to continue . . .
using ...
namespace OldTimes
{
0 references | 0 authors | 0 changes
class Program
{
private static int counter = 0;
private const int loopcount = 1000000000;
2 references | 0 authors | 0 changes
static void ThreadMain()
{
for (int i = 0; i < loopcount; i++)
{
counter++;
}
}
0 references | 0 authors | 0 changes
static void Main(string[] args)
{
Thread t1 = new Thread(new ThreadStart(ThreadMain));
Thread t2 = new Thread(new ThreadStart(ThreadMain));
t1.Start();
t2.Start();
t1.Join();
t2.Join();
Console.WriteLine("counter == {0}",counter);
Console.WriteLine("Should be {0}",2*loopcount);
}
}
}

Semaphore



- Fundamental synchronization primitive
- Two basic functions (on a semaphore s)
  - s.P()
    - May block in case semaphore already "occupied"
  - s.V()
    - Never blocks; releases semaphore and may free a blocked thread
- Dijkstra (1968), "THE" Multiprogramming System
  - P = passeren (request entry)
  - V = vrygeven (release)

Semantic

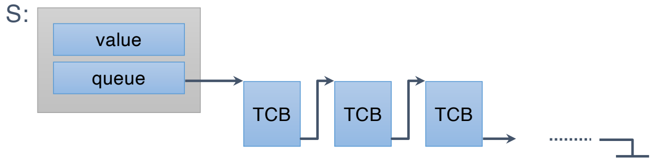
- Semaphores are counting signals
  - s.P(): Thread waits for some signal s
  - s.V(): Thread sends a signal s
- Semaphor = Integer (s.value)
  - Initialized with a s.value ≥ 0

```
void P () {
  s.value--;
  if (s.value < 0)
    Block calling thread;
}
```

Of course, P and V themselves are critical sections and must be protected!

```
void V () {
  s.value++;
  if (s.value < 1)
    Put a thread waiting in s into ready queue;
}
```

Implementation



P

- Execution of P must be atomic
- Thread k calls P

```
s.value--;
if (s.value < 0) {
   tail(s.queue) :=
k;
   block(k);
   }
```

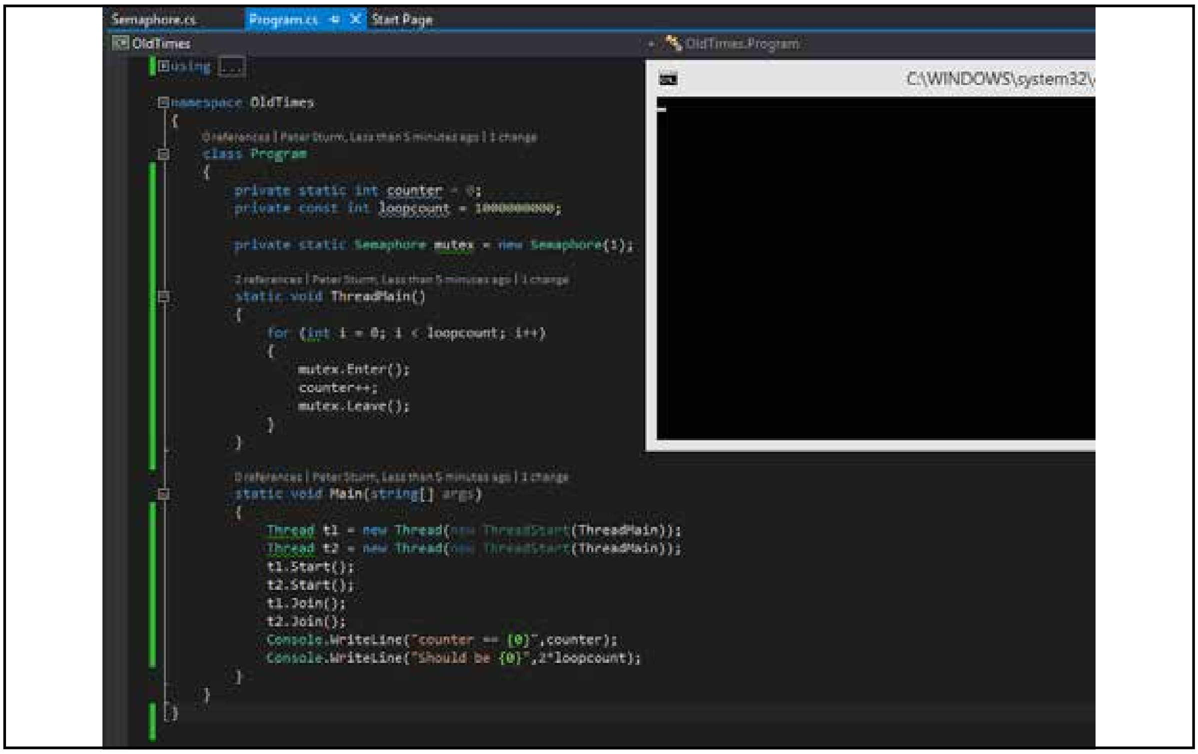
V

- Execution of V must be atomic
- Calling thread normally doesn' t block

```
s.value++;
if (s.value <= 0) {
   ready(head(s.queue))
   delhead(s.queue)
   }
```

Semaphor

```
class Program
  {
    private static int counter = 0;
    private const int loopcount = 1000000000;

    private static Semaphore mutex = new Semaphore(1);

    static void ThreadMain()
    {
        for (int i = 0; i < loopcount; i++)
        {
            mutex.Enter();
            counter++;
            mutex.Leave();
        }
    }

    static void Main(string[] args)
    {
        Thread t1 = new Thread(new ThreadStart(ThreadMain));
        Thread t2 = new Thread(new ThreadStart(ThreadMain));
        t1.Start();
        t2.Start();
        t1.Join();
        t2.Join();
        Console.WriteLine("counter == {0}",counter);
        Console.WriteLine("Should be {0}",2*loopcount);
    }
  }
```

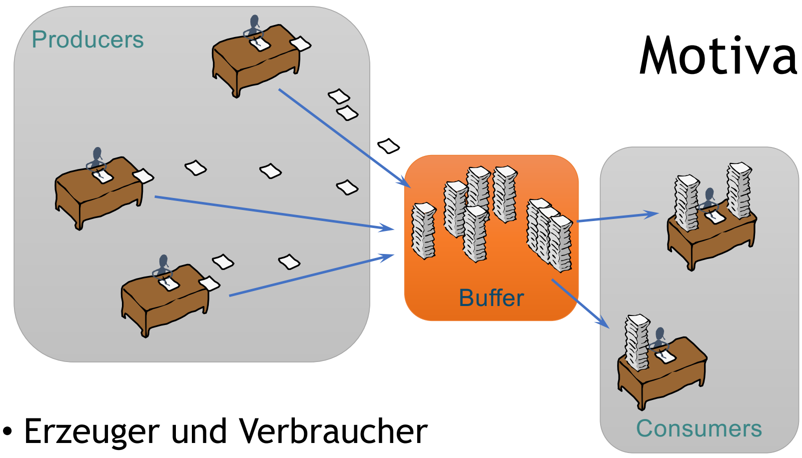



```
class Program
    {
        private static int counter = 0;
        private const int loopcount = 1000000000;

        static void ThreadMain()
        {
            for (int i = 0; i < loopcount; i++)
            {
                Interlocked.Increment(ref counter);
            }
        }

        static void Main(string[] args)
        {
            Thread t1 = new Thread(new ThreadStart(ThreadMain));
            Thread t2 = new Thread(new ThreadStart(ThreadMain));
            t1.Start();
            t2.Start();
            t1.Join();
            t2.Join();
            Console.WriteLine("counter == {0}",counter);
            Console.WriteLine("Should be {0}",2*loopcount);
        }
    }
```

Interlocked

Producer Consumer

Motivation



- Erzeuger und Verbraucher
- Puffer fester Größe
  - Geschwindigkeitsunterschiede
- Wie löst man das Problem?

Outline of Buffer

```
public class Buffer
{
    public Buffer ( int n )
    {
    }

    /// Inserts another good into the buffer.
    /// May block until free space is available.
    public void Produce ( int good )
    {
    }

    /// Consumes a good stored inside the buffer.
    /// May signal blocked producer threads.
    public int Consume ()
    {
        return 42;
    }
}
```

Producer (C#)

```
public class Producer
    {
        public Producer ( Buffer b )
        {
            buffer = b;
            my_id = this.GetHashCode();
            ThreadStart ts = new ThreadStart(Run);
            my_thread = new Thread(ts);
            my_thread.Start();
        }

        private void Run ()
        {
            Console.WriteLine("Producer {0}: started ...",my_id);
            int good = this.GetHashCode() * 1000000;
            while (true)
            {
                buffer.Produce(good);
                Console.WriteLine("Producer {0}: good {1} stored",my_id,good);
                good++;
            }
        }

        private Buffer buffer;
        private Thread my_thread;
        private int my_id;
    }
```

Consumer (C#)

```
public class Consumer
     {
          public Consumer ( Buffer b )
          {
               buffer = b;
               my_id = this.GetHashCode();
               ThreadStart ts = new ThreadStart(Run);
               my_thread = new Thread(ts);
               my_thread.Start();
          }

          private void Run ()
          {
               Console.WriteLine("Consumer {0}: started ...",my_id);
               while (true)
               {
                    int good = buffer.Consume();
                    Console.WriteLine("Consumer {0}: good {1} retrieved",my_id,good);
               }
          }

          private Buffer buffer;
          private Thread my_thread;
          private int my_id;
     }
```

The Buffer (C#)

```
public class Buffer
     {
          public Buffer ( int n )
          {
               this.n = n;
               slots = new int[n];
               mutex = new Semaphore(1);
               slots_available = new Semaphore(n);
               goods_available = new Semaphore(0);
          }

          …

          private int n;
          private int [] slots;
          private int free = 0;
          private int used = 0;
          private Semaphore mutex;
          private Semaphore slots_available;
          private Semaphore goods_available;
     }
```

```
public void Produce ( int good )
{
    slots_available.P();
    mutex.P();
    slots[free] = good;
    free = (free+1) % n;
    mutex.V();
    goods_available.V();
}

public int Consume ()
{
    goods_available.P();
    mutex.P();
    int good = slots[used];
    used = (used+1) % n;
    mutex.V();
    slots_available.V();
    return good;
}
```

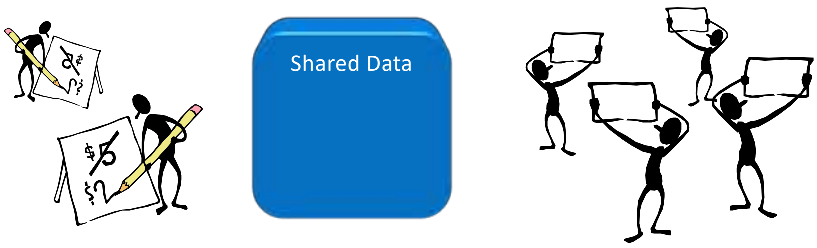
The Optimal Buffer (C#)

```
public class Buffer
    {
        public Buffer ( int n )
        {
            this.n = n;
            slots = new int[n];
            mutex_p = new Semaphore(1);
            mutex_c = new Semaphore(1);
            slots_available = new Semaphore(n);
            goods_available = new Semaphore(0);
        }

        …

        private int n;
        private int [] slots;
        private int free = 0;
        private int used = 0;
        private Semaphore mutex_p, mutex_c;
        private Semaphore slots_available;
        private Semaphore goods_available;
    }
```

```
public void Produce ( int good )
{
    slots_available.P();
    mutex_p.P();
    slots[free] = good;
    free = (free+1) % n;
    mutex_p.V();
    goods_available.V();
}

public int Consume ()
{
    goods_available.P();
    mutex_c.P();
    int good = slots[used];
    used = (used+1) % n;
    mutex_c.V();
    slots_available.V();
    return good;
}
```

Reader/writer

The Problem



- A shared data structure is used by multiple threads
- Writer threads
  - These threads modify the shared data and require mutual exclusion
- Reader threads
  - Since readers don't modify data, they can access the shared data structure simultaneously

Starting Point: Mutual Exclusion

```
Semaphore Sanctum = new Semaphore(1);
```

Shared Data

```
while (true) {
  Sanctum.P();
  // Change data
  Sanctum.V();
}
```

Writer

```
while (true) {
  Sanctum.P();
  // Read data
  Sanctum.V();
}
```

Reader

Reader Preference (Faulty)

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
int readers_inside = 0;
```

Writer

```
while (true) {
  Sanctum.P();
  // Change data
  Sanctum.V();
}
```

Reader

```
while (true) {
  if (readers_inside == 0)
    Sanctum.P();
  readers_inside++;
  // Read data
  readers_inside--;
  if (readers_inside == 0)
    Sanctum.V();
}
```

Why Does It Fail?

Reader

```
while (true) {
  if (readers_inside == 0)
    Sanctum.P();
  readers_inside++;
  // Read data
  readers_inside--;
  if (readers_inside == 0)
    Sanctum.V();
}
```

- Multiple readers may access the shared variable readers_inside concurrently
- Testing and setting variable must be atomic among readers
- Mutual exclusion required

2. Reader Preference (Correct)

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
Semaphore RMutex = new Semaphore(1);
int readers_inside = 0;
```

Writer

```
while (true) {
  Sanctum.P();
  // Change data
  Sanctum.V();
}
```

Reader

```
while (true) {
  RMutex.P();
  if (readers_inside == 0)
    Sanctum.P();
  readers_inside++;
  RMutex.V();
  // Read data
  RMutex.P();
  readers_inside--;
  if (readers_inside == 0)
    Sanctum.V();
  RMutex.V();
}
```

Reader/Writer (Prio Writer)

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
Semaphore RMutex = new Semaphore(1);
Semaphore WMutex = new Semaphore(1);
Semaphore PreferWriter = new Semaphore(1);
Semaphore ReaderQueue = new Semaphore(1);
int readers_inside = 0;
int writers_interested = 0;
```

Writer

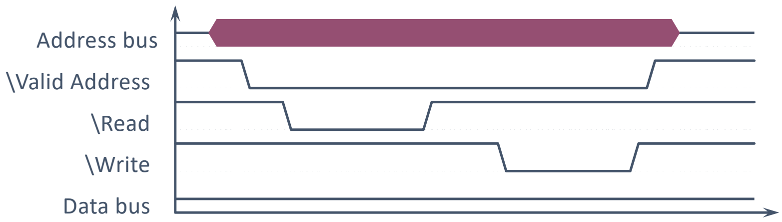
```
while (true) {
  WMutex.Enter();
  if (writers_interested == 0)
    PreferWriter.Enter();
  writers_interested++;
  WMutex.Leave();
  Sanctum.Enter();
  // Change data
  Sanctum.Leave();
  WMutex.Enter();
  writers_interested--;
  if (writers_interested == 0)
    PreferWriter.Leave();
  WMutex.Leave();
}
```

Reader

```
while (true) {
  ReaderQueue.Enter();
  PreferWriter.Enter();
  RMutex.Enter();
  if (readers_inside == 0)
    Sanctum.Enter();
  readers_inside++;
  RMutex.Leave();
  PreferWriter.Leave();
  ReaderQueue.Leave();
  // Read data
  RMutex.Enter();
  readers_inside--;
  if (readers_inside == 0)
    Sanctum.Leave();
  RMutex.Leave();
}
```

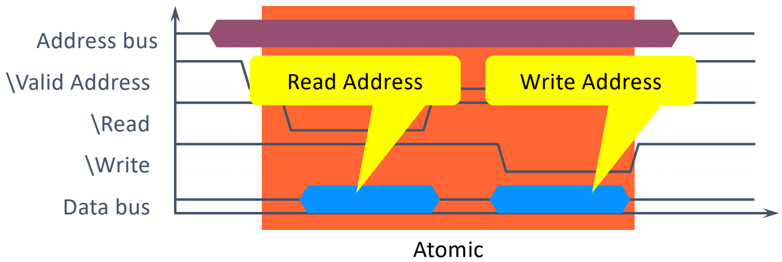

Spin locks

Atomic Read and Write Cycle



- Combined read and write cycle to a single address
- Not interruptable even on a multiprocessor system
- All modern CPUs offer such instructions
  - SPARC: LDSTUB (Atomic Load-Store Unsigned Byte), SWAP
  - 680x0: TAS (Test and Set)
- No privileged instructions required for synchronization

Atomic Read and Write Cycle



- Combined read and write cycle to a single address
- Not interruptable even on a multiprocessor system
- All modern CPUs offer such instructions
  - SPARC: LDSTUB (Atomic Load-Store Unsigned Byte), SWAP
  - 680x0: TAS (Test and Set)
- No privileged instructions required for synchronization

Example Using Atomic "Test and Set"

- h = TAS Address
  - h := Memory[Address]
  - Memory[Address] := 1

Thread 1:

```
…
Loop: h = TAS mutex
      if (h) goto Loop
// mutex is now 1

// Critical section

mutex = 0;
```

Thread n:

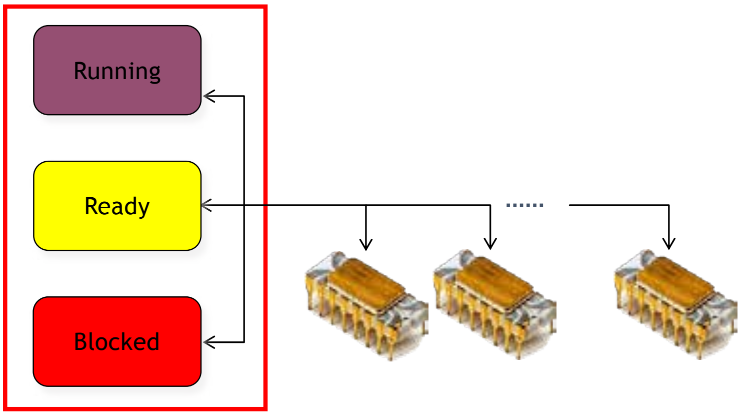
```
…
Loop: h = TAS mutex
      if (h) goto Loop
// mutex is now 1

// Critical section

mutex = 0;
```

Spinlock

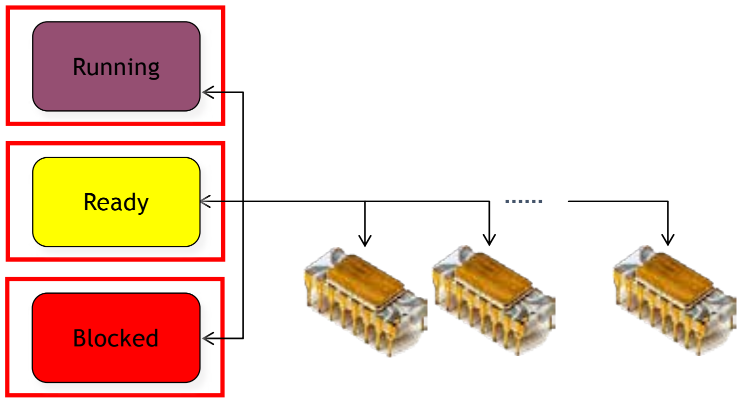
Where to use?

- Useful only for short waiting time
  - Accessing the scheduling queues on a ManyCore CPU
  - Synchronizing on a hardware event happening in the near future
- Potential risks
  - Starvation, Livelocks, …
  - Performance with respect to multiprocessors and caching

Spinlocks



Spinlocks



User-Land Spinlocks?

- ManyCores
- Kooperative Anwendungen
  - Gleichzeitig aktive Threads
  - Enge Interaktion über Speicher
- OS-API
- Managed
  - C#: System.Threading.Spinlock

Immer wichtiger!

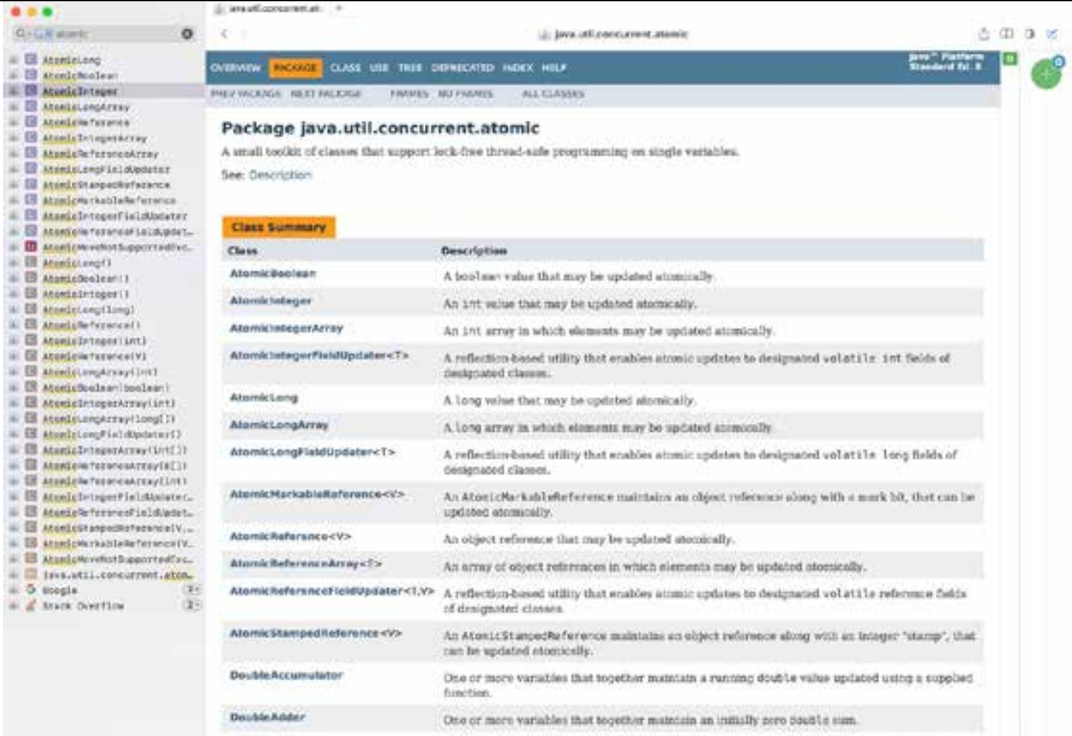
- Schnelle Kommunikation bei kooperierenden Threads
- Gleichzeitige Ausführung
  - Gang Scheduling

.NET Support

- Class Interlocked in System.Threading
  - static int Increment (ref int x)
    - Increments variable x atomically and returns result
  - static int Decrement (ref int x)
    - Decrements variable x atomically and returns result
  - static int Exchange (ref int x, int y)
    - Assigns value y to variable x atomically and retuns the previous value
  - static int CompareExchange (ref int x, int y, int c)
    - Assigns value y to x atomically iff x == c; returns previous value of x

Spinlock Example in .NET

```
class Spinlock
{
    public void Enter ()
    {
        int v;
        do
        {
            v = Interlocked.CompareExchange(ref mutex,1,0);
        } while (v == 1);
    }

    public void Leave ()
    {
        Interlocked.Exchange(ref mutex,0);
    }

    private int mutex = 0; // 0 means nobody inside
}
```



Package java.util.concurrent.atomic

A small toolkit of classes that support lock-free thread-safe programming on single variables.

See: Description

Class Summary

Class	Description
AtomicBoolean	A boolean value that may be updated atomically.
AtomicInteger	An int value that may be updated atomically.
AtomicIntegerArray	An int array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.
AtomicLong	A long value that may be updated atomically.
AtomicLongArray	A long array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.
AtomicMarkableReference<V>	An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.
AtomicStampedReference<V>	An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.
DoubleAccumulator	One or more variables that together maintain a running double value updated using a supplied function.
DoubleAdder	One or more variables that together maintain an initially zero double sum.

AtomicInteger

float	floatValue() Returns the value of this AtomicInteger as a float after a widening primitive conversion.
int	get() Gets the current value.
int	getAndAccumulate(int x, IntBinaryOperator accumulatorFunction) Atomically updates the current value with the results of applying the given function to the current and given values, returning the previous value.
int	getAndAdd(int delta) Atomically adds the given value to the current value.
int	getAndDecrement() Atomically decrements by one the current value.
int	getAndIncrement() Atomically increments by one the current value.
int	getAndSet(int newValue) Atomically sets to the given value and returns the old value.
int	getAndUpdate(IntUnaryOperator updateFunction) Atomically updates the current value with the results of applying the given function, returning the previous value.
int	incrementAndGet() Atomically increments by one the current value.
int	intValue() Returns the value of this AtomicInteger as an int.
void	lazySet(int newValue)

Critical Section without Help

Critical Section

- A “sensible” sequence of instructions which may only be executed by one thread at a time
- Mutual exclusion required inside critical section
- Must be defined by the application
- Bracketing by some **`Enter()`** and **`Leave()`** operations:

Critical Section

```
…
EnterCriticalSection();
Sensible instruction 1;
…
Sensible instruction k;
LeaveCriticalSection();
…
```

Correct Implementation

- Mutual exclusion
  - At any given time at most one thread is inside
- No deadlocks
  - Threads are delayed for a finite time only before entering
- Fairness
  - Any thread who wants to enter will enter eventually
- Efficiency
  - Threads are not delayed needlessly
  - Threads currently not interested shouldn’t do superfluous work

Basic Program Structure

- We assume 2 threads (id 0 and 1) continuously entering the critical section:

```
public void Loop ()
{
  DateTime start = DateTime.Now;
  do
  {
    EnterCriticalSection();
    int c = Interlocked.Increment(ref threads_inside_critical_section);
    if (c > 1)
    {
      Console.WriteLine("Oops. More than one thread inside critical section");
    }
    Thread.Sleep(0); // Forces context switch
    Interlocked.Decrement(threads_inside_critical_section);
    LeaveCriticalSection();
    } while (((TimeSpan)(DateTime.Now - start)).TotalSeconds < seconds_to_run);
}
```

Different Implementations

- Do not make use of any synchronization support
  - No primitives provided by the operating system
  - No hardware support

Volatile Data Structures

- Hardware (CPU, MMU), runtime support, and compiler normally relax memory consistency to improve performance
  - Re-odering memory access as long as program semantics are not altered
  - Don’t expect memory to change by side-effects
  - Normally, a single thread is assumed
- Unexpected behavior in multi-threaded environments
- Data structures that are accessed concurrently by multiple threads should be marked **`volatile`**
  - Nobody will cache these data
  - Compiler expects these data to change asynchronously
  - Every access will read the memory address

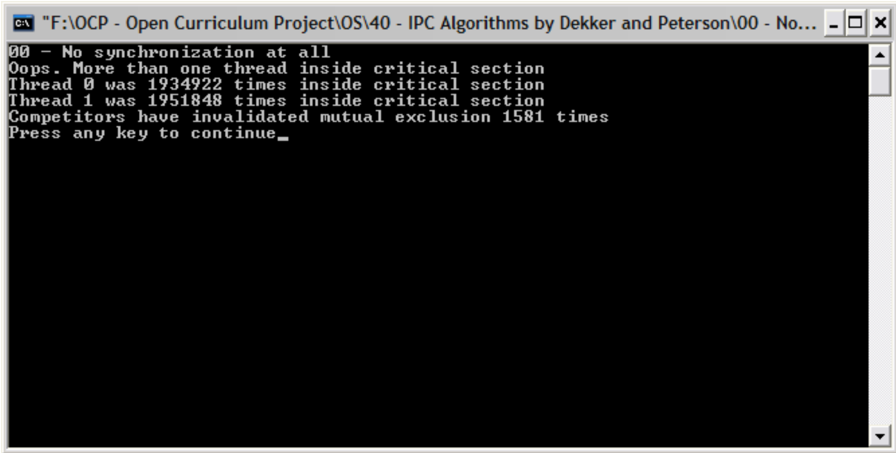
Volatile and C#

- Specific keyword
  - `volatile bool flag`
- Specific static methods in Thread available
  - `object Thread.VolatileRead ( ref object )`
  - `Thread.VolatileWrite ( ref object, object)`
  - Various overloads
- The problem with arrays:
  - Consider "`volatile bool [] flags` …"
    - Reference to array is marked volatile
    - The booleans themselves are not!
    - `Thread.VolatileRead()` and `Thread.VolatileWrite()` don't work on boolean
  - Workaround: Integers instead of boolean
    - 1 ⇔ true
    - Some subsequent examples may look strange ☹

Starting Point

- Empty functions to enter and leave the critical section
- No concurrency control
  - Several threads may be inside critical section in parallel

Result



Step 1: Alternating Token

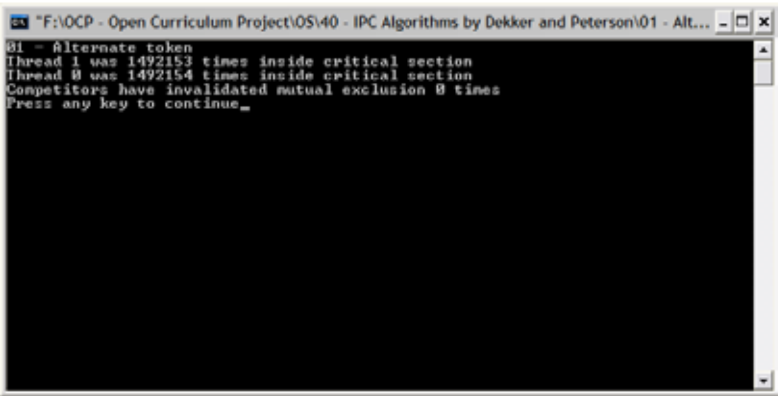
- Implementing a token-based approach
  - The thread who owns the token may enter, the other waits
  - The token must circulate among the participating threads

Alternating Token

- For each thread
  - self = own thread, rival = rival thread
- Solution

```
private volatile static int turn = 0;

public void EnterCriticalSection ()
{
  while (turn != self) /* Busy Wait */;
}

public void LeaveCriticalSection ()
{
  turn = rival;
}
```

Result



- No invalidation
- Times inside critical section identical (±1)

Comments

- Mutual exclusion is guaranteed
- It is a fair solution
  - Provided any
- It is a non-blocking solution (Busy Waiting)
  - Requires preemptive scheduling
- Threads must pass the token actively although they are not interested in entering the critical section
  - Solution doesn't consider different interest and enter frequency of all threads involved

Step 2: Someone Inside

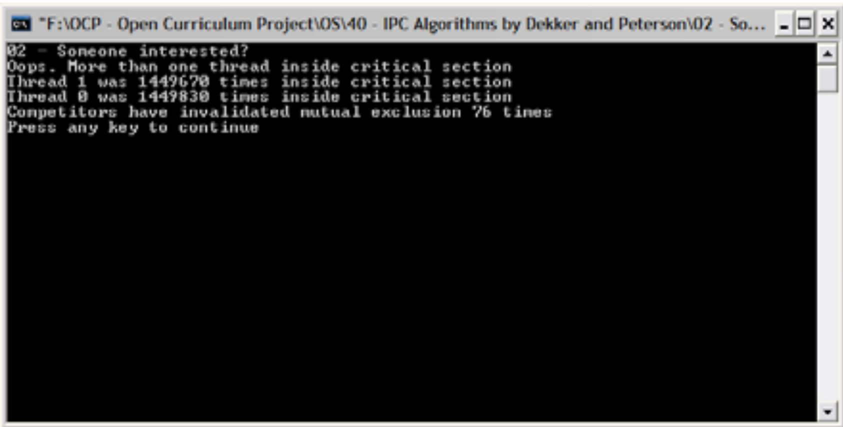
- Threads set flag when inside critical section
  - Only enter if the rival is not already inside

Someone About to Enter

- For each thread
  - self = own thread, rival = rival thread
- Solution

```
private static bool[] inside = new int[2] { 0, 0 };

public void EnterCriticalSection ()
{
  while (inside[rival] == 1) /* Busy Wait */ ;
  inside[self] = 1;
}

public void LeaveCriticalSection ()
{
  inside[self] = 0;
}
```

Result



- Mutual exclusion condition still invalidated several times

Comments

- Mutual Exclusion not guaranteed

```
Thread 0:
inside[0] == 0;

...
while (inside[1] == 1) ;

inside[0] = 1;

// Inside Critical Section
```

Context Switch

```
Thread 1:
inside[1] == 0;

...
while (inside[0] == 1) ;


inside[1] = 1;

// Inside Critical Section
```

- Testing and setting flag is not atomar
- Gets worse with increasing number of threads

Step 3: Set before Test

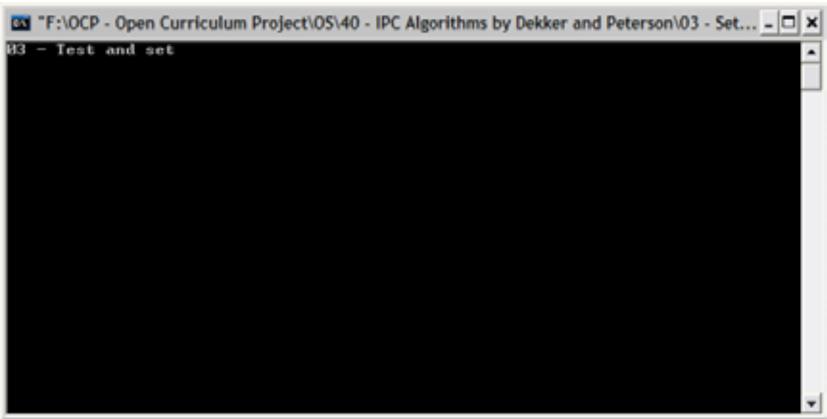
- Threads express their interest before entering
  - Only enter if the rival is not interested too

Set Before Test

- For each thread
  - self = own thread, rival = rival thread
- Solution

```
private static bool[] interested = new int[2] { 0, 0 };

public void EnterCriticalSection ()
{
  interested[self] = 1;
  while (interested[rival] == 1) /* Busy Wait */ ;
}

public void LeaveCriticalSection ()
{
  interested[self] = 0;
}
```

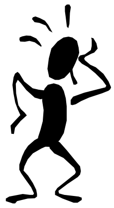
Result



- No progress after 1 minute! Why?

Comments

- Mutual exclusion guaranteed
  - At most one thread (actually 0) inside
- Livelock



```
Thread 0:

interested[0] = 1;
while (interested[1]==1)
;
while (interested[1]==1)
;
while (interested[1]==1)
;
while (interested[1]==1)
;
while (interested[1]==1)
;
...
```

Context Switch

```
Thread 1:

interested[1] = 1;




while (interested[0]==1)
;
while (interested[0]==1)
;
while (interested[0]==1)
;
while (interested[0]==1)
```

Step 4: I want, I don’t want, I want …

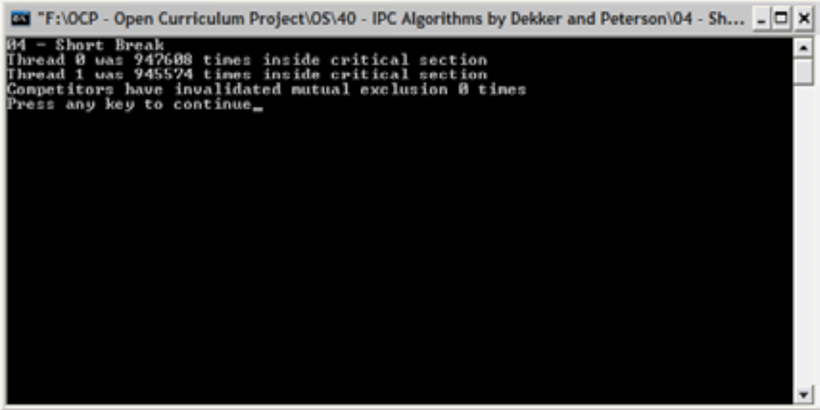
- Trying to avoid livelock
  - In case of imminent livelock release flag for a short period of time and try again
  - Well, it’s still busy waiting, but …

Short Break

- For each thread
  - self = own thread, rival = rival thread
- Solution

```
private static bool[] interested = new int[2] { 0, 0 };

public void EnterCriticalSection ()
{
  interested[self] = 1;
  while (interested[rival] == 1) {
    interested[self] = 0;
    /* Short Break */;
    interested[self] = 1;
     }
}

public void LeaveCriticalSection ()
{
  interested[self] = 0;
}
```

Result



- Seems to work?

Comments

- Mutual exclusion guaranteed
- Mutual Courtesy
  - Threads in lockstep can exhibit livelock-like behavior

```
Thread 0:

interested[0] = 1;
while (interested[1]==1)
  interested[0] = 0;
  /* Short Break */
  interested[0] = 1;
while (interested[1]==1)
  interested[0] = 0;
  /* Short Break */
  interested[0] = 1;
...c
```

```
Thread 1:

interested[1] = 1;
while (interested[0]==1)
  interested[1] = 0;
  /* Short Break */
  interested[1] = 1;
while (interested[0]==1)
  interested[1] = 0;
  /* Short Break */
  interested[1] = 1;
...
```

The Algorithm by Dekker

- Combination of versions 4 and 2
  - 4: Set before test, but avoid livelocks by releasing the own flag again
  - 2: Alternating token to resolve conflicts fair

Dekker: Source Code

```
bool [] interested = new bool[2] { false, false };
int turn = 0;

EnterCriticalSection () {
  interested[self] = true;
  while (interested[rival]) {
    if (turn == rival) {
      interested[self] = false;
      while (turn == rival) /* Busy Wait */ ;
      interested[self] = true;
    }
  }
}

LeaveCriticalSection () {
  turn = rival;
  interested[self] = false;
}
```

The Algorithm by Peterson

- Improvement of Dekkers' algorithm
- Conflict is resolved through race condition
  - Who made the last write to turn?

Peterson: Source code

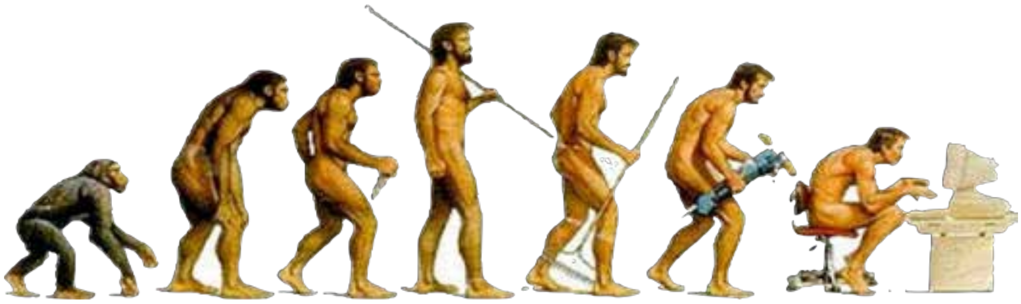
```
bool [] interested = new bool[2] { false, false };
int turn = 0;

EnterCriticalSection () {
  interested[self] = true;
  turn = rival; // Volatile race condition
  while (interested[rival] and (turn == rival)) {
    /* Busy Wait */ ;
  }
}

LeaveCriticalSection () {
  interested[self] = false;
}
```

If it hurts
Stop doing it ☺

Paul Butcher

zeit für veränderungen!

... Warten?



Parallelisierende Compiler

- SIMD-artige Probleme (Number Crunching)
- High Performance Fortran (HPF)
  - Erweiterung von Fortran 90
  - FORALL
- Datenflußanalyse
  - Forschungsgebiet
  - Alternative Rechnerarchitekturen

parallel loop

primzahlen

```
static bool is_prime(int v)
{
    if (v < 2) return false;
    int d = 2;
    while (d * d <= v)
    {
        if (v % d == 0) return false;
        d = d == 2 ? 3 : d + 2;
    }
    return true;
}
```

sequentiell

```
// Sequential loop
DateTime start = DateTime.Now;
for (int c = 0; c < limit; c++)
    primes[c] = is_prime(c);
TimeSpan m = DateTime.Now - start;
Console.WriteLine("Serial 0 ... {0} needs {1} ms", limit, m.TotalMilliseconds);
```

parallel

```
// Parallel loop
start = DateTime.Now;
Parallel.For(0, limit, c => { primes[c] = Program.is_prime(c); });
m = DateTime.Now - start;
Console.WriteLine("Parallel 0 ... {0} needs {1} ms", limit, m.TotalMilliseconds);
```

Stolpersteine

- Willkürliche Abarbeitungsreihenfolge
- Concurrency-safe?
  - Datenabhängigkeiten über Iterationen hinweg!!!
  - Seiteneffekte

have a break

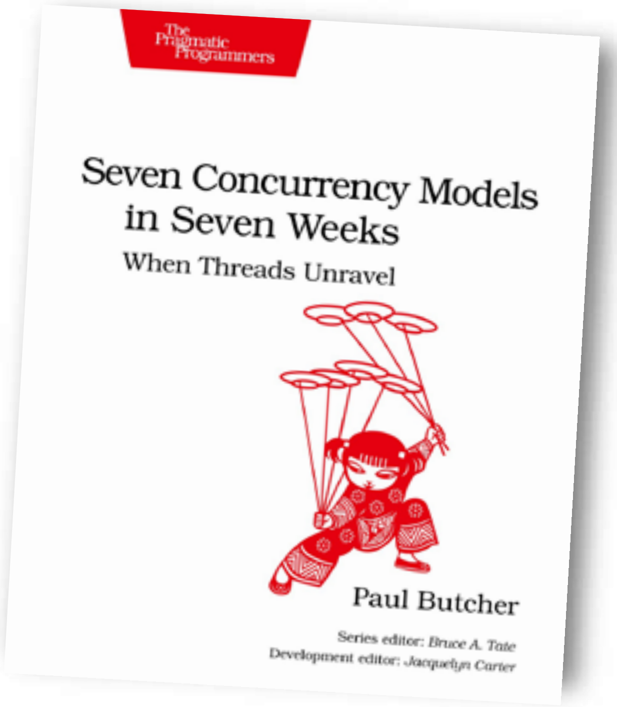
- Mehrere Iterationen gleichzeitig aktiv
- Parallel Break
  - Alle Iterationen mit kleinerem Index werden noch beendet
- Parallel Stop
  - Schnurzpiepegal

beispiel

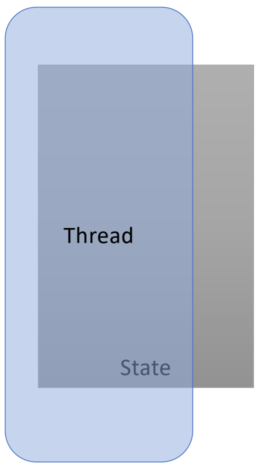
```
bool[] state = new bool[n];
for (int i = 0; i < n; i++) state[i] = false;
Parallel.For(0, n, (l, loopstate) => {
    if (l != n / 2) BurnTime() else loopstate.Break();
    state[l] = true;
});
for (int i = 0; i < n; i++) Console.Write("{0}", state[i] ? 'T' : 'F');
```

es gibt noch mehr ...

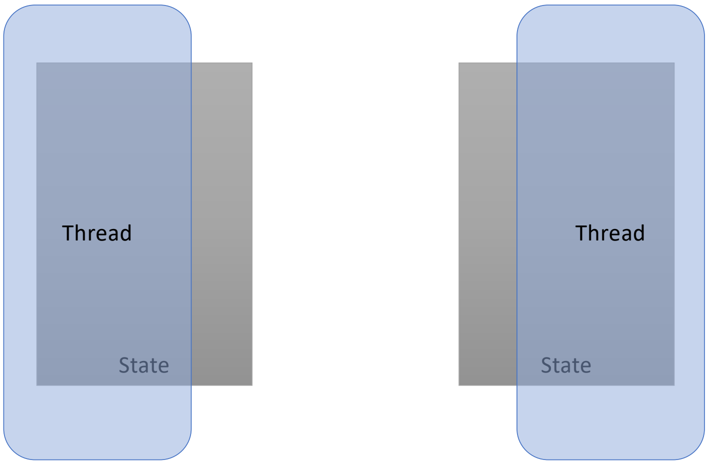
- External Loop Cancellation
- Granularität steuern
  - Partitioner
- Minimale und maximale Threadanzahl spezifizierbar



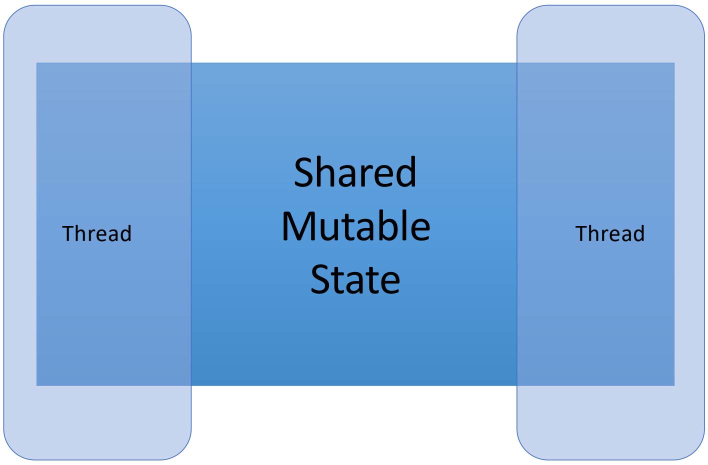
single-threaded

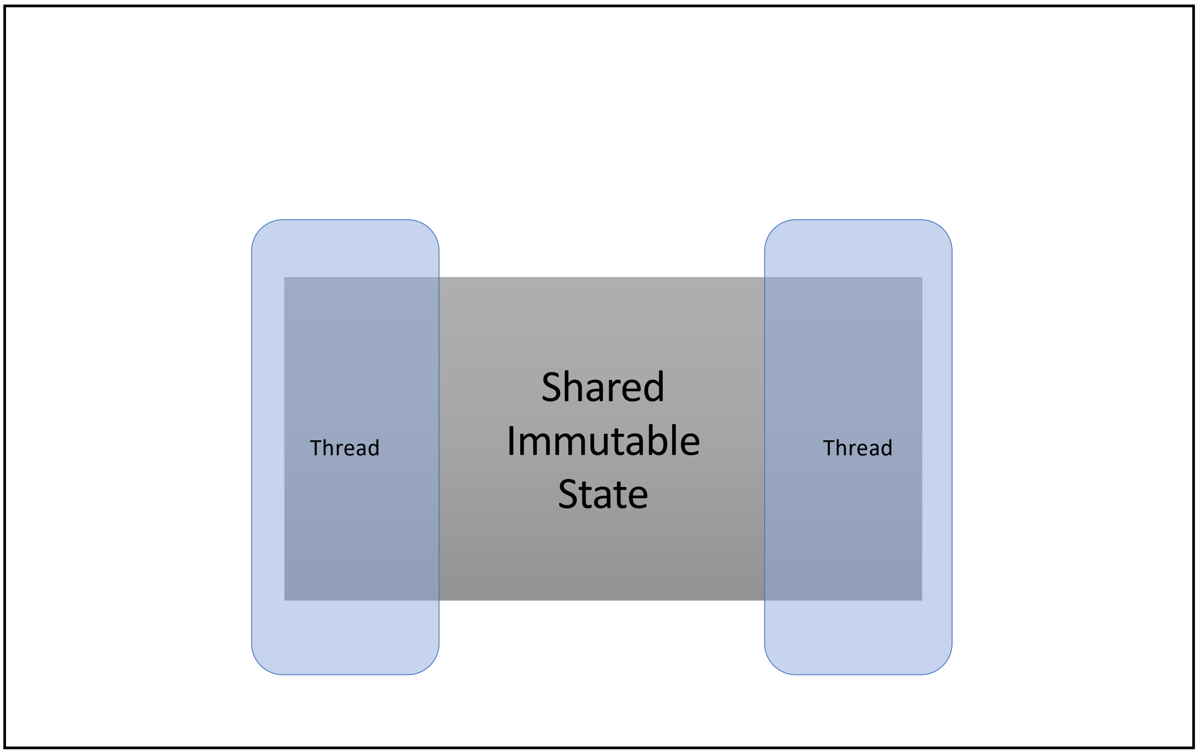


multi-process



Multi-threaded



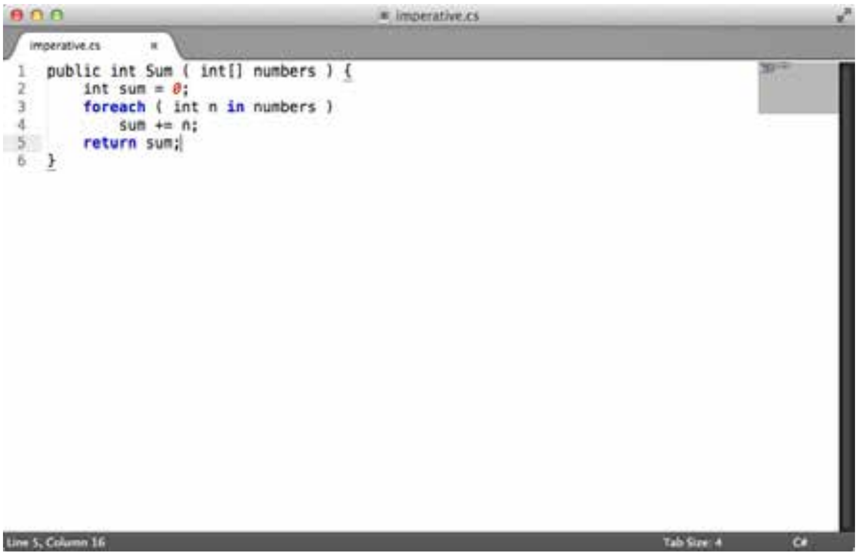

Thread
Shared Immutable State
Thread


immutable State?

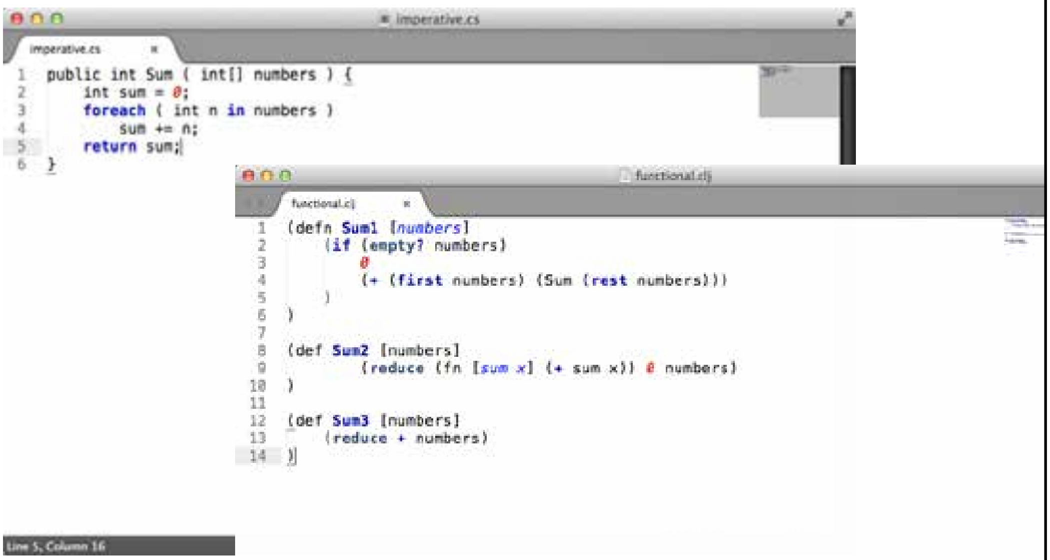
FP

Functional Programming

imperativ



funktional


imperative.cs
public int Sum (int[] numbers) {
int sum = 0;
foreach (int n in numbers)
sum += n;
return sum;
}
Line 5, Column 16
functional.clj
(defn Sum1 [numbers]
(if (empty? numbers)
0
(+ (first numbers) (Sum (rest numbers)))
)
)
(def Sum2 [numbers]
(reduce (fn [sum x] (+ sum x)) 0 numbers)
)
(def Sum3 [numbers]
(reduce + numbers)
)

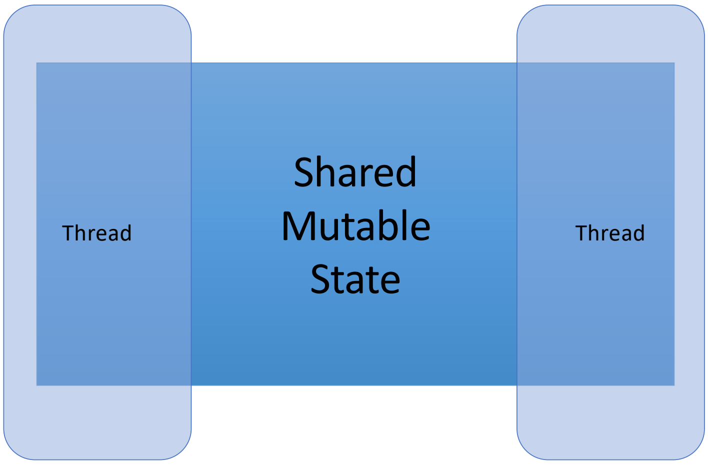

Farming-Simulator 2013
CASE IH
600

CLOJURE

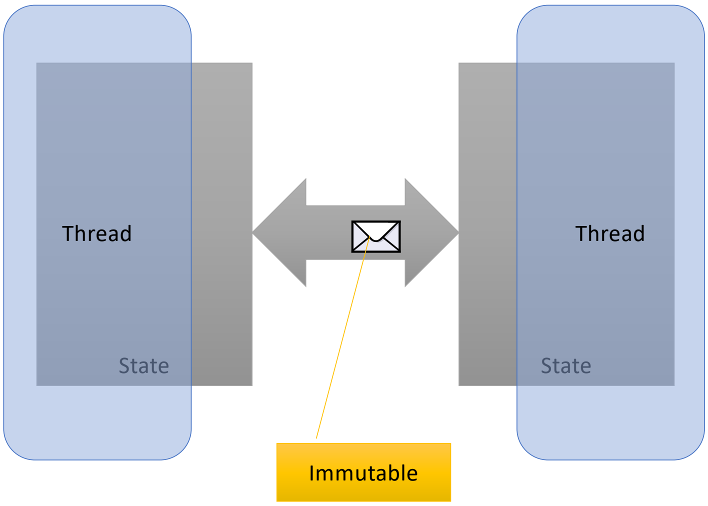
impure functional language

- „Concurrency-aware mutable variables“
  - Imperativ: Mutable ist Standardfall
- Persistenz
  - Transaktions-artiges Fortschreiben des Objektzustands
  - Vorheriger Zustand während der Änderung zugreifbar
- Software-Transactional Memory

Multi-threaded

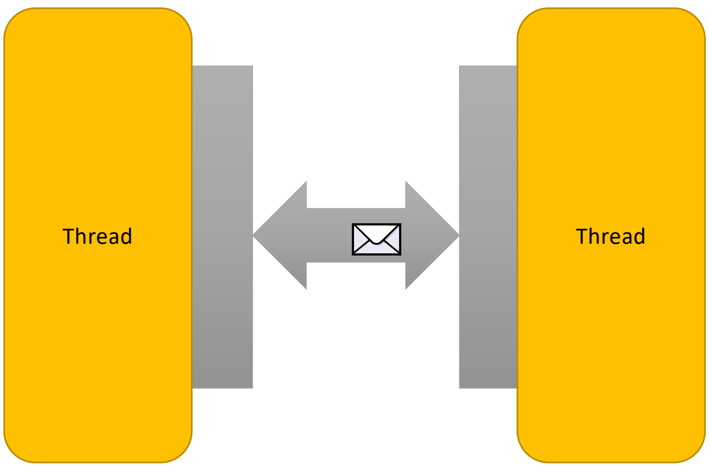


Messages

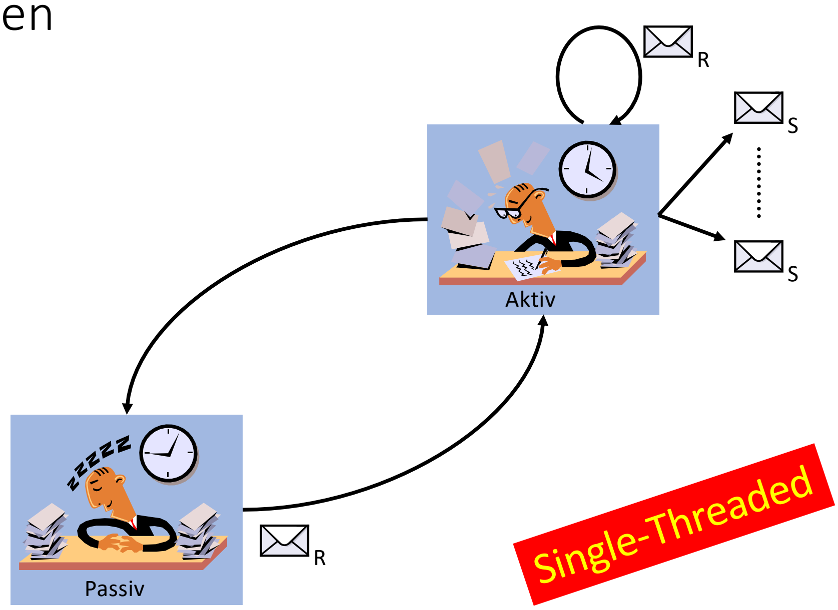


Actors

fokus SubjeKt: Actors



Aktoren



Elixir

- Läuft auf der „Erlang Virtual Machine" (BEAM)
- Erlang
  - Funktionale Sprache (Ericsson)
  - OTP (Open Telecom Platform)

ØMQ

If there’s one lesson we’ve learned from 30+ years of concurrent programming, it is:

Just don’t share state!

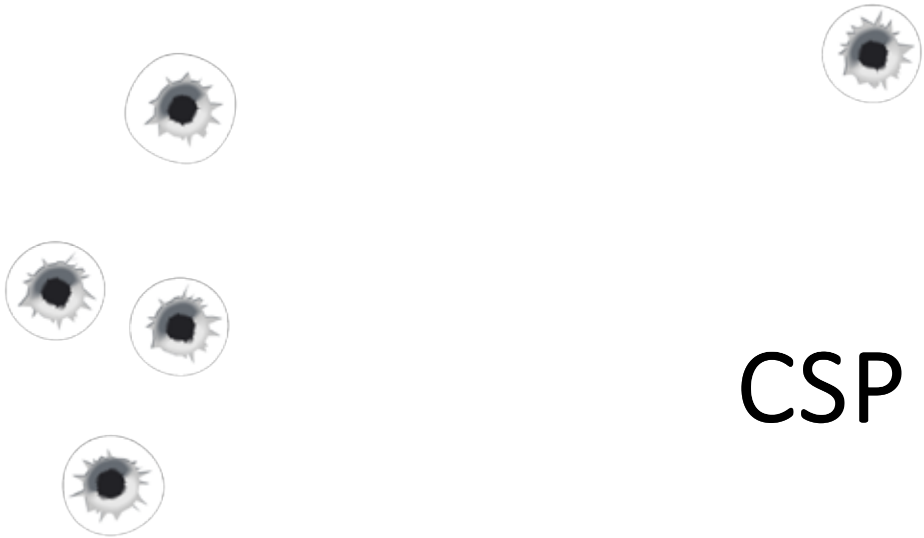
Pieter Hintjens

Ansatz

- Kein „Shared State“ zwischen Threads
- Austausch von Nachrichten
  - inproc-Transport
- Support für Thread-Signaling
  - PAIR sockets

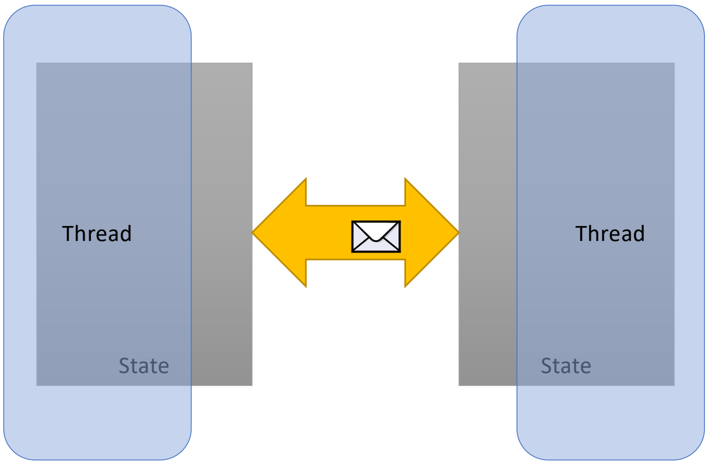
TPL Data Flow

- Actor-artiger Ansatz in .NET
  - Zusätzliches Package
- Blocktypen
  - Buffer
  - TransformBlock (DataIn, DataOut)
  - Broadcast Block
  - JoinBlock
  - ...

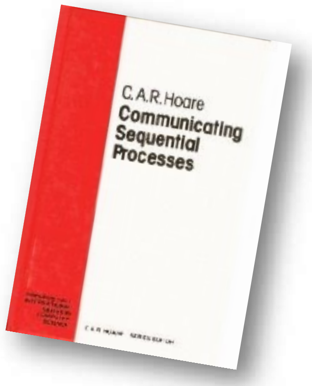


CSP

FOKUS Message: CSP



CSP



- Communicating Sequential Processes
  - Hoare, Process Calculus, 1978
- Channels
- Go
- Clojure, core.async

Java Streams (?)

- Vergleichbar .NET TPL
- Stream = Sequenz von Elementen (Objekten)
  - Lazy
- Sources
  - Collections, Arrays, I/O, ...
- Data Processing Operations
  - Filter, Map, Reduce, Find, Match, ...
- Terminals
  - Collect, Count, ForEach, ...

Skalierbarkeit

- Traditionelle DBMS
  - Vertikale Skalierung

- NoSQL-Systeme
  - Horizontale Skalierung

Nicht im Buch! ☺

APM

Asynchronous Programming Model

futures, Promises und mehr

Asynchron

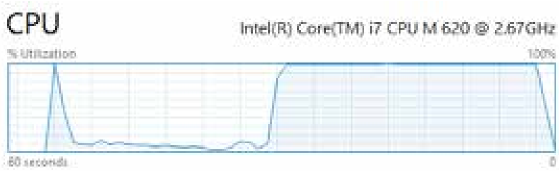
- Blockierende Aufrufe meiden
- Ursprung im Bereich „User Interface“
- Ausgangspunkt für nebenläufige Programmierung

.NET

- Verfeinerung und Ausbau von Task, Task<T>
  - Task-Ketten
  - ...
- Thread Pool
- await
  - Synchrone Kapselung eines asynchronen Aufrufs
  - Compiler

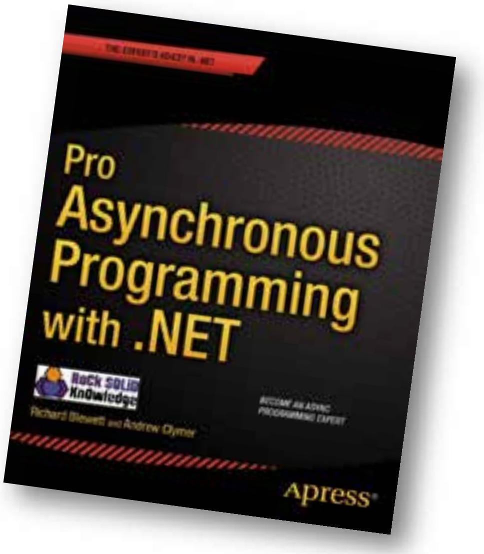
beispiel

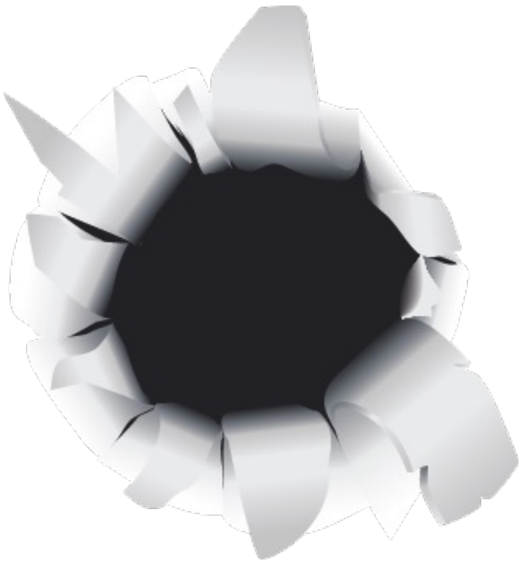
```
long nPrimes = 0;
for (long k = 1; k < 2000000; k++)
{
    Task<bool> aPrime = Task.Factory.StartNew<bool>(() => Primes.IsPrime(k));
    if (aPrime.Result) nPrimes++;
}
Console.WriteLine("{0} primes found",nPrimes);
```



Bemerkung zu .NET

- Nur Einzelaspekte angesprochen
- Concurrent Collections
- Monitor
- ...





Fazit