



# Multi Threading



- » Skan.ai - chief Architect
- » Ai.robotics - chief Architect
- » Genpact - solution Architect
- » Welldoc - chief Architect
- » Microsoft
- » Mercedes
- » Siemens
- » Honeywell



Mubarak

# Agenda

- Concurrency
  - Expectations
- Locks
  - Years of Exp
- Best Practices
  - Technology stack
- Anti Patterns
- Debugging



Mubarak

- Implicit
  - Main Thread
  - Green Thread
  - OS Thread

# Parallelism

## Task Parallelism

- Thread pool thread
  - Executor
  - ForkJoin (divide and conquer)
- Dedicated Thread
  - Background Thread (long running)
  - Foreground Thread (must complete)

## Data parallelism

- Declarative (Parallel Streams)
- Partitioning ?
- Collation ?
- Imperative Thread Pool
- ForkJoin (divide and conquer)
- Executor

# Parallelism

## Task Parallelism

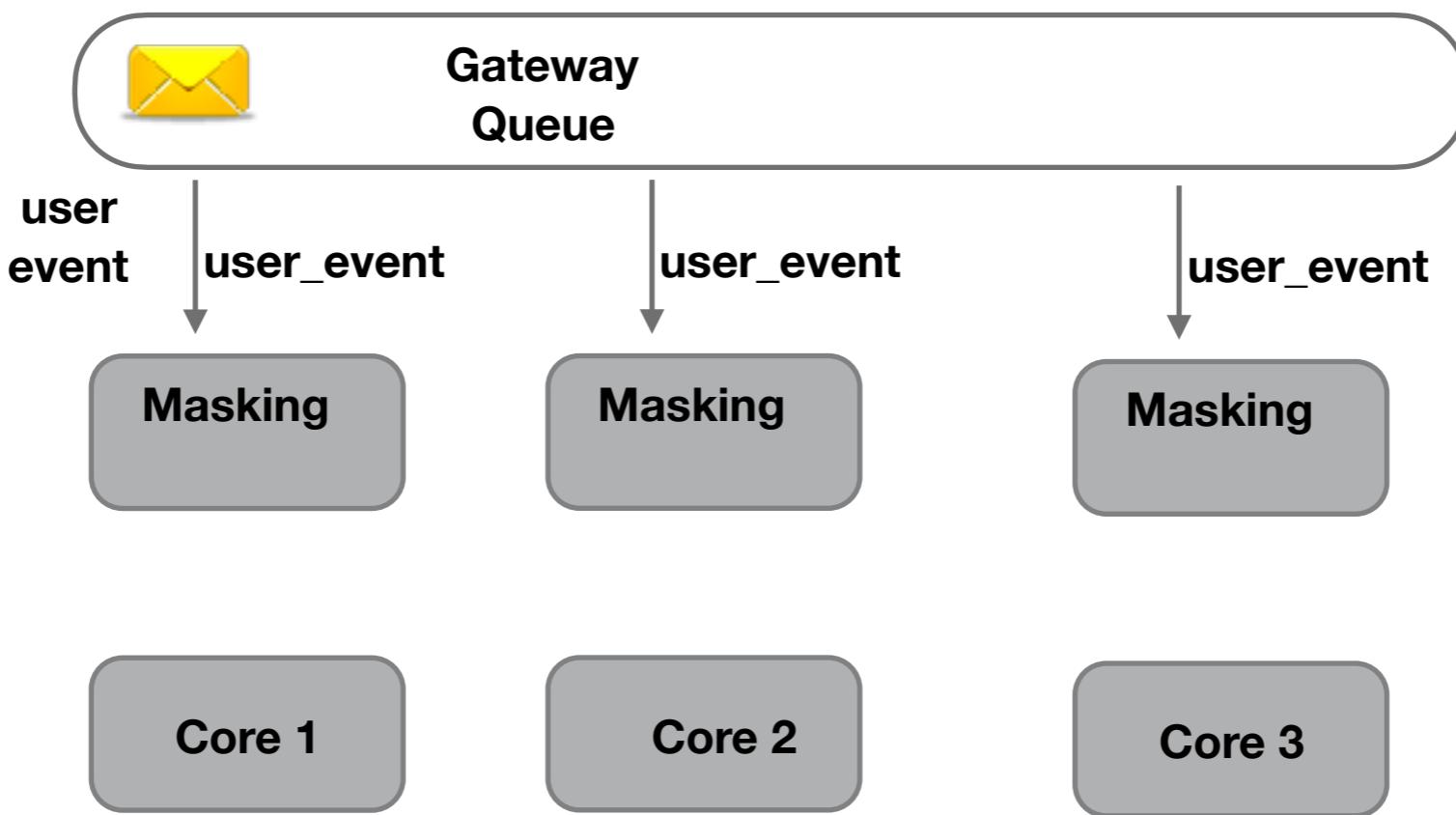
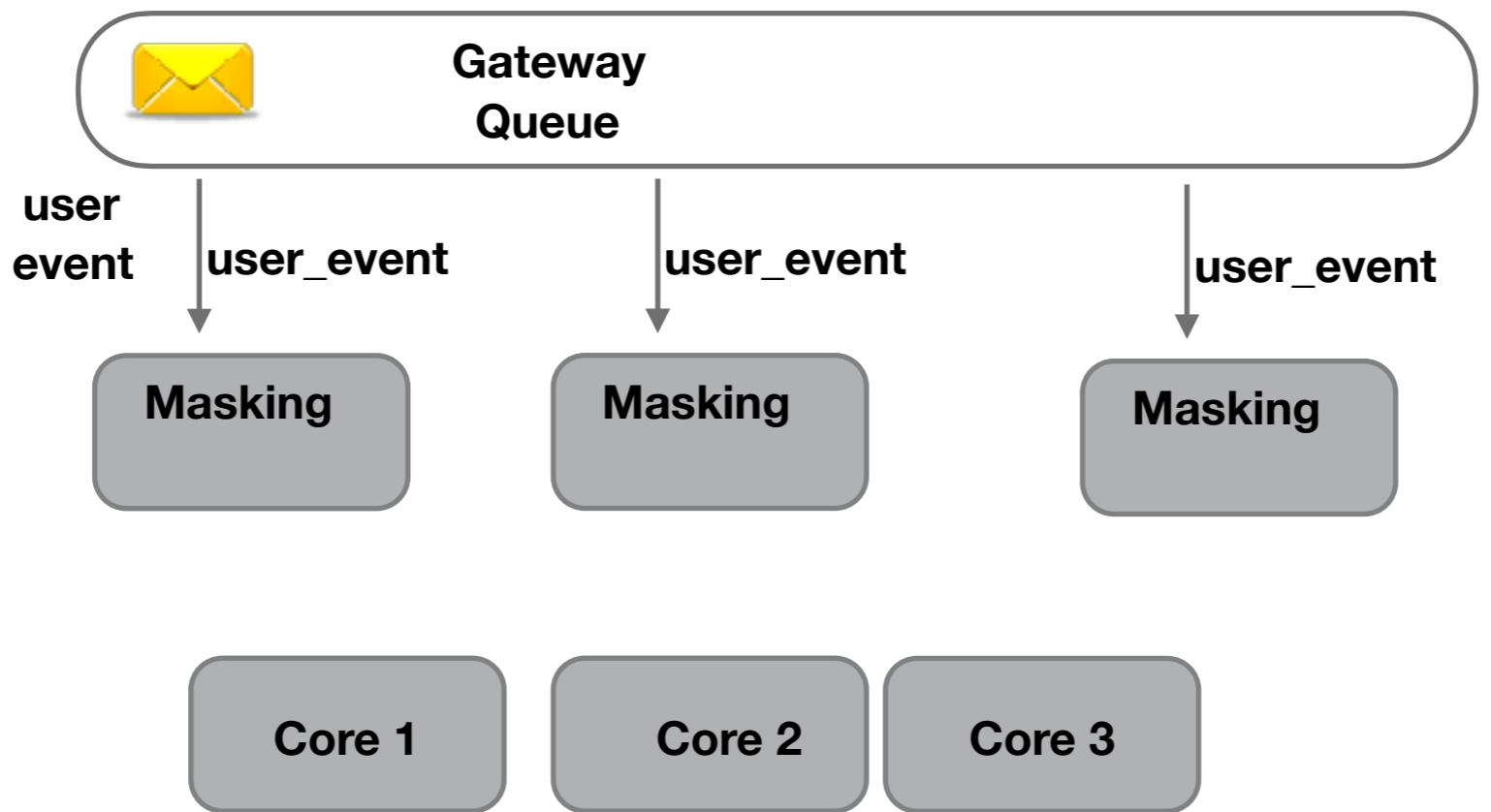
- Thread pool
- Dedicated Thread
  - Background Thread
  - Foreground Thread

## Data parallelism

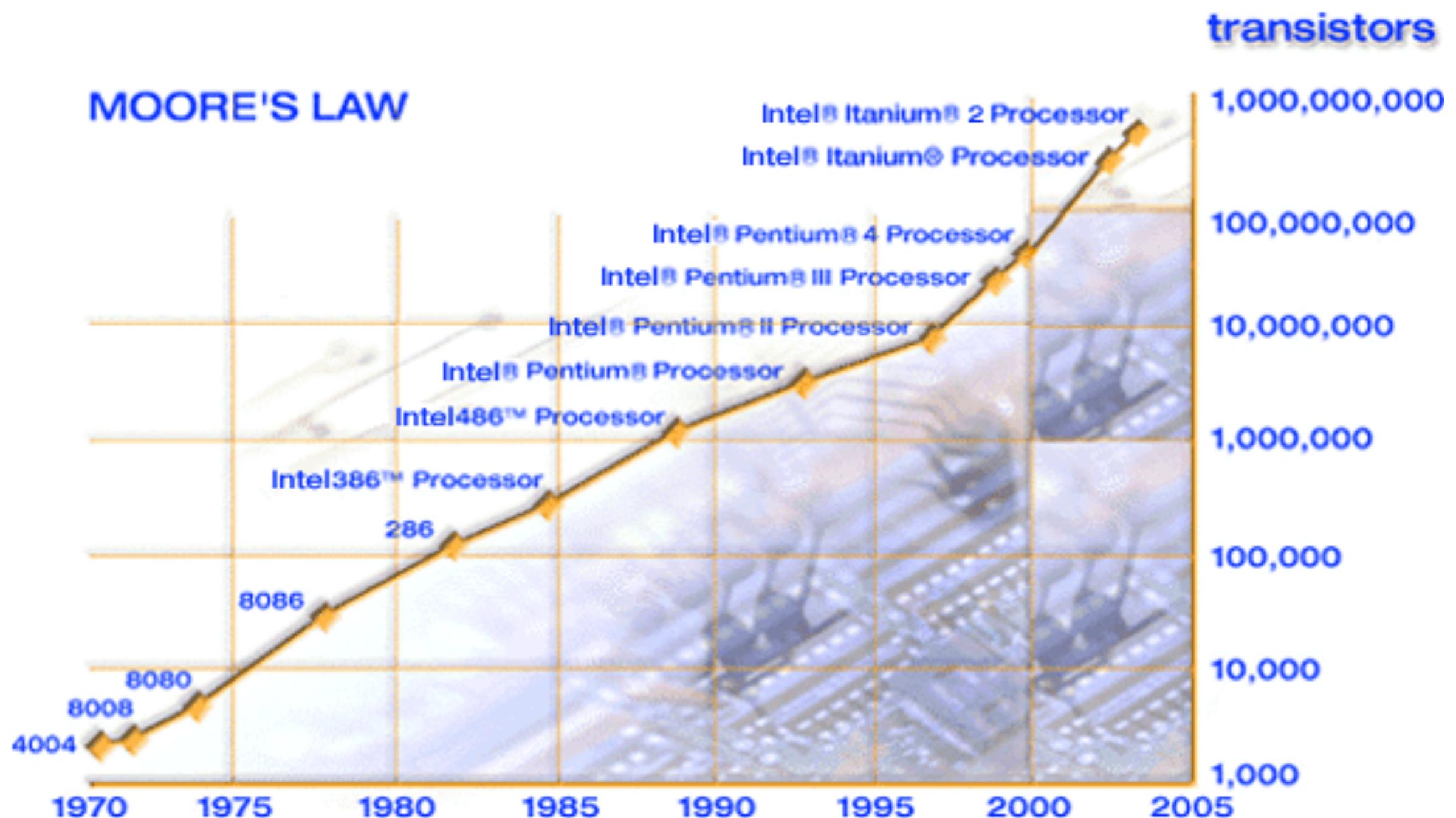
- Declarative

## IO parallelism

- Async IO
- Non Blocking IO
- Blocking IO



# The Free Lunch is Over



Unfortunately adding transistors stopped translating into faster processors, primarily because of power consumption and the heat generated.

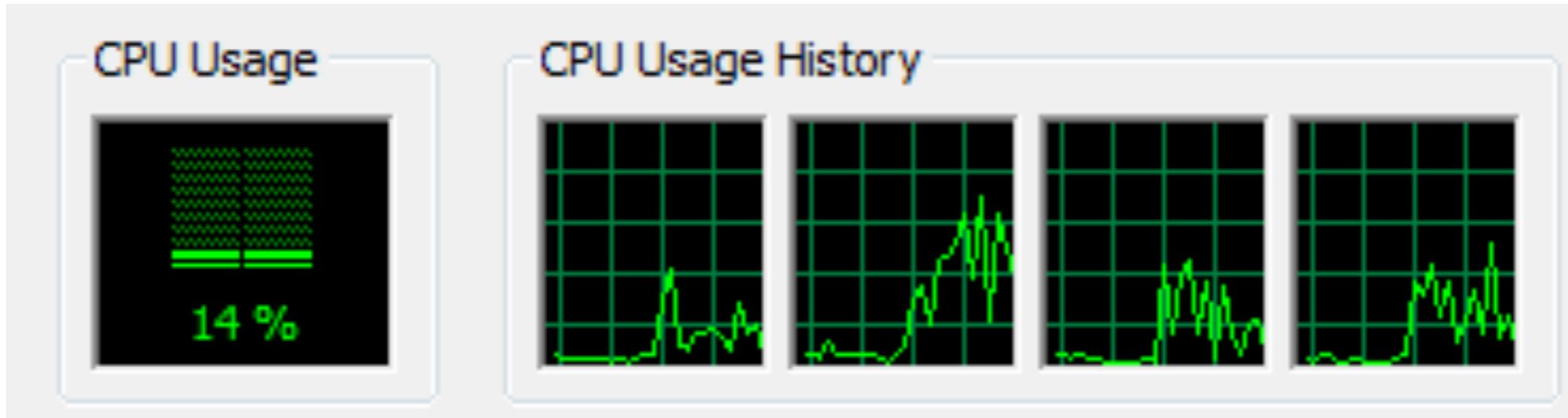
# Example

```
private static void doWork(int instance)
{
    double result = Math.Acos(new Random().NextDouble()) *
Math.Atan2(new Random().NextDouble(), new Random().NextDouble());

    for (int i = 0; i < 20000; i++)
    {
        result += (Math.Cos(new Random().NextDouble()) *
            Math.Acos(new Random().NextDouble()));
    }
}
```

# Single-Threaded Programming

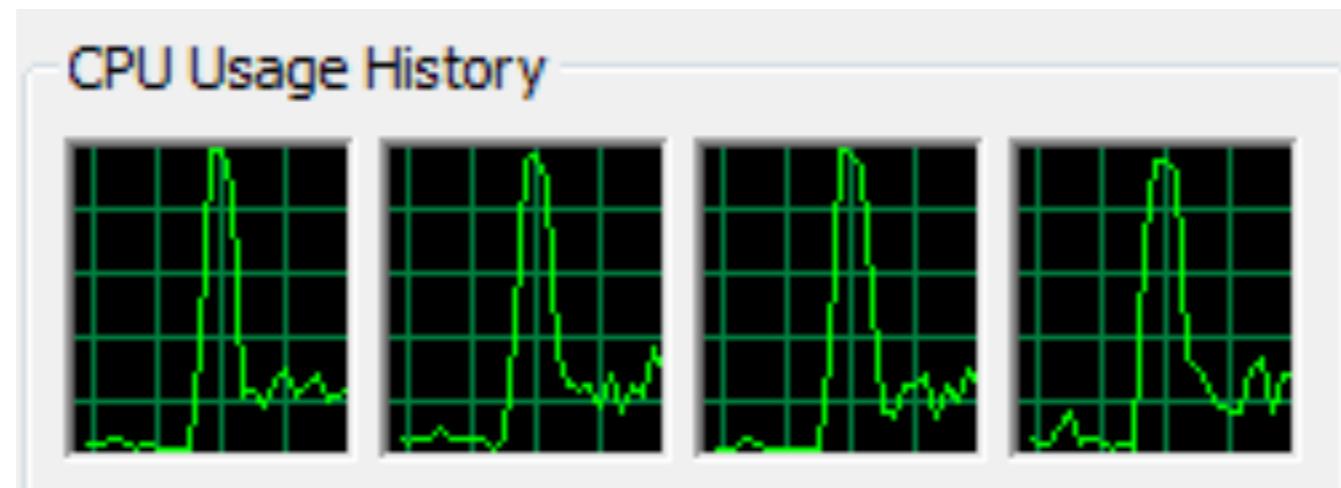
```
for (int i = 0; i < 100; i++)  
{  
    doWork(i);  
}
```



With four cores : over 12 seconds

# Parallel Programming

```
//for (int i = 0; i < 100; i++)  
  
Parallel.For(0, 100,  
    delegate(int i){      doWork(i);          } );
```



The process finishes in 3.7 seconds instead of 12.

# Parallel Programming

Individual tasks  
may take a  
little longer

```
0 took 145 milliseconds
1 took 124 milliseconds
2 took 121 milliseconds
3 took 125 milliseconds
4 took 122 milliseconds
5 took 124 milliseconds
6 took 122 milliseconds
7 took 123 milliseconds
8 took 121 milliseconds
9 took 121 milliseconds
10 took 122 milliseconds
11 took 124 milliseconds
12 took 123 milliseconds
13 took 122 milliseconds
14 took 122 milliseconds
```

Single-threaded

```
4 took 176 milliseconds
5 took 173 milliseconds
3 took 198 milliseconds
2 took 261 milliseconds
6 took 246 milliseconds
7 took 286 milliseconds
1 took 355 milliseconds
0 took 385 milliseconds
8 took 195 milliseconds
12 took 210 milliseconds
16 took 231 milliseconds
10 took 329 milliseconds
20 took 174 milliseconds
18 took 244 milliseconds
```

Parallel

some as high as  
400-  
milliseconds

**Java version 5 was targeted to parallel execution  
enabled by multi-core CPU**

# **Foreground Threads**

**vs**

# **Background Threads**



Try to avoid using foreground threads  
as much as possible



You should use foreground threads to execute tasks that you really want to complete, like flushing data from a memory buffer out to disk.

**x + y -> 3 cpu cycles**

**fun(x,y) -> 10 cpu cycle**

**CreateThread -> ?**

# Cost of a Memory

- For x86 CPU, the thread's context uses about 700 bytes of memory.
- For x64 the context is about 1,240 bytes of memory.
- For IA64 CPUs, the context is about 2,500 bytes of memory



# Cost of a Thread

- 200,000 cycles to create a thread
- 100,000 to destroy one
- context switch costs 2,000-8,000 cycles



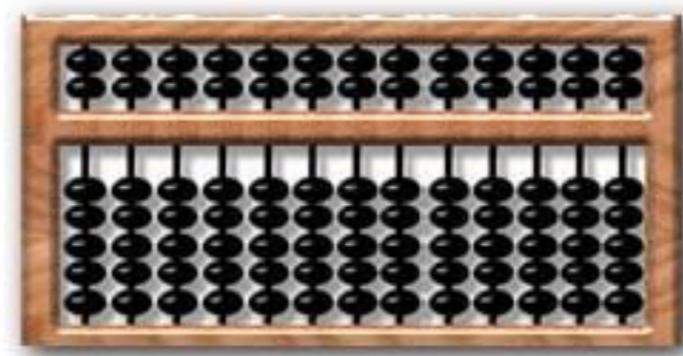
# **Don't create Threads**



User threads as implemented by Java virtual machines are called green threads

# Total Threads

- A 32-bit process has 2 GB.
- After Win32 DLLs load, the CLR DLLs load, the native heap and the managed heap is allocated, there is approximately 1.5 GB left.
- Each thread requires more than 1 MB of memory for its user-mode stack and thread environment block (TEB)
- The most threads you can get in a 32-bit process is about 1,360.





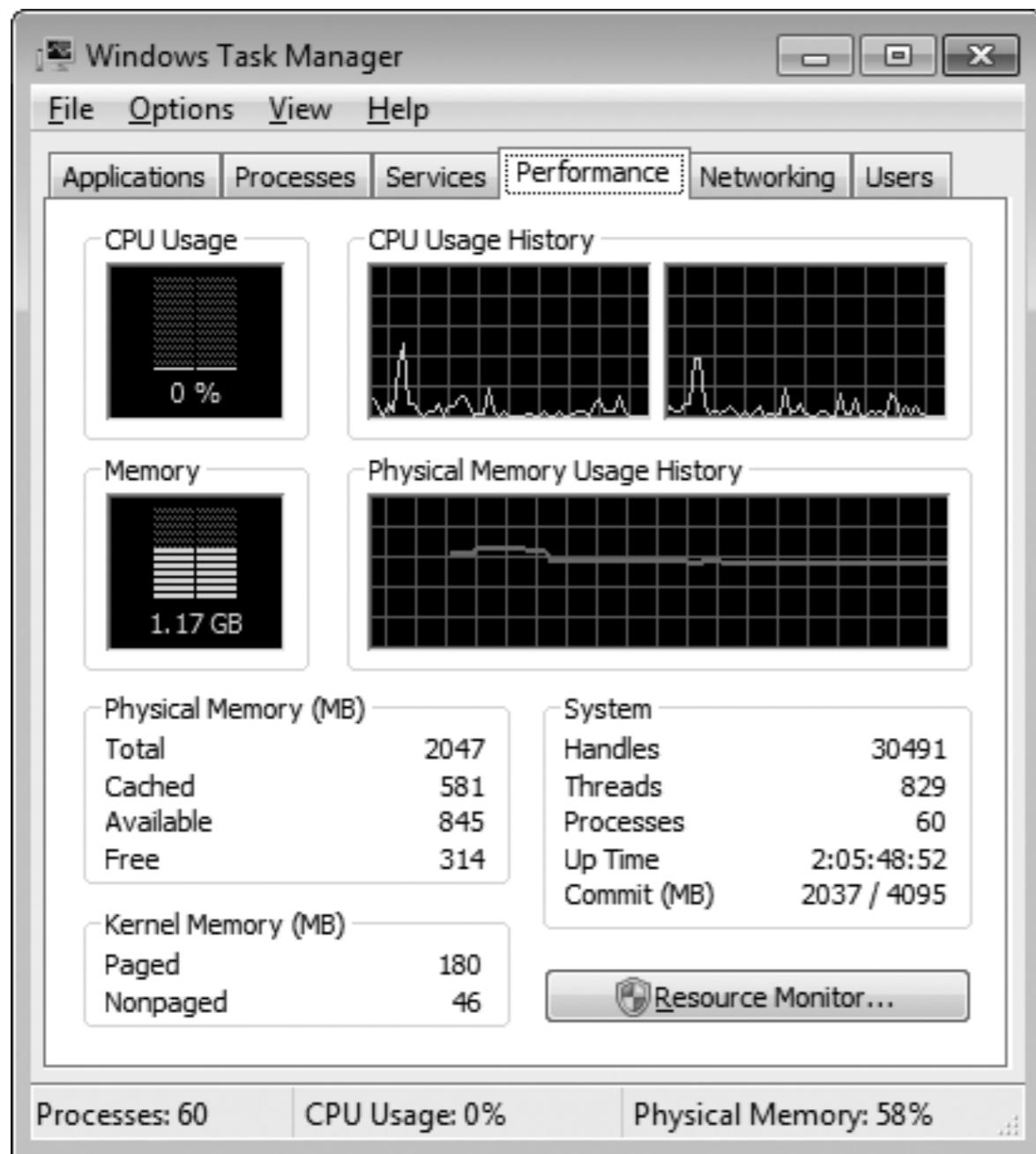
What is the Ideal Number of Threads ?



# CARPOOLING

Because Gas is Just Too Expensive

# Do these applications need all these threads



Windows Task Manager

File Options View Help

Applications Processes Services Performance Networking Users

Image Name	CPU	Memory (...	Threads	Description
OUTLOOK.EXE *32	00	86,392 K	38	Microsoft ...
devenv.exe *32	00	98,048 K	34	Microsoft ...
msnmsgr.exe *32	00	28,612 K	34	Windows ...
explorer.exe	00	24,336 K	32	Windows ...
iexplore.exe *32	00	44,388 K	21	Internet ...
WINWORD.EXE *32	00	34,888 K	13	Microsoft ...
MagicMouse.exe	00	7,944 K	13	
wlcomm.exe *32	00	6,208 K	11	Windows ...
csrss.exe	00	3,216 K	11	
iexplore.exe *32	00	3,716 K	10	Internet ...
msvsmon.exe	00	6,124 K	10	Visual Stu...
PushPin.exe	01	4,564 K	10	PushPin
MagicMouse.exe	00	3,948 K	10	
mspaint.exe	00	18,792 K	9	Paint

Show processes from all users

End Process

Processes: 60 CPU Usage: 9% Physical Memory: 59%

This screenshot shows the Windows Task Manager's Processes tab. It displays a list of running processes with the following columns: Image Name, CPU, Memory (partial view), Threads, and Description. The processes listed include OUTLOOK.EXE, devenv.exe, msnmsgr.exe, explorer.exe, iexplore.exe, WINWORD.EXE, MagicMouse.exe, wlcomm.exe, csrss.exe, iexplore.exe, msvsmon.exe, PushPin.exe, MagicMouse.exe, and mspaint.exe. At the bottom, there are buttons for 'Show processes from all users' and 'End Process'. Status bars at the bottom show 60 processes, 9% CPU usage, and 59% physical memory.

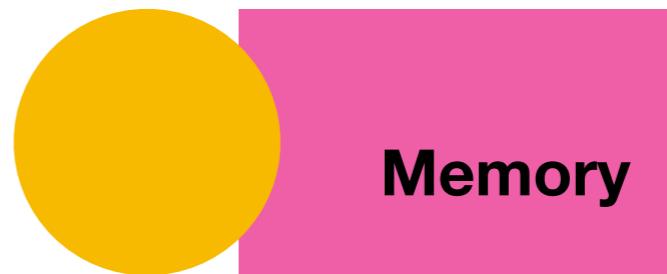


Playing with thread pool limits usually results in making an application perform worse, not better

## Max : 5 threads

executor.execute()  
Memory reader

executor.execute()  
Memory writer



executor.execute()  
executor.execute()  
executor.execute()  
executor.execute()  
executor.execute()  
executor.execute()

# When To Create Thread



Do you require a foreground thread. Thread pool threads are always background threads.

# When To Create Thread



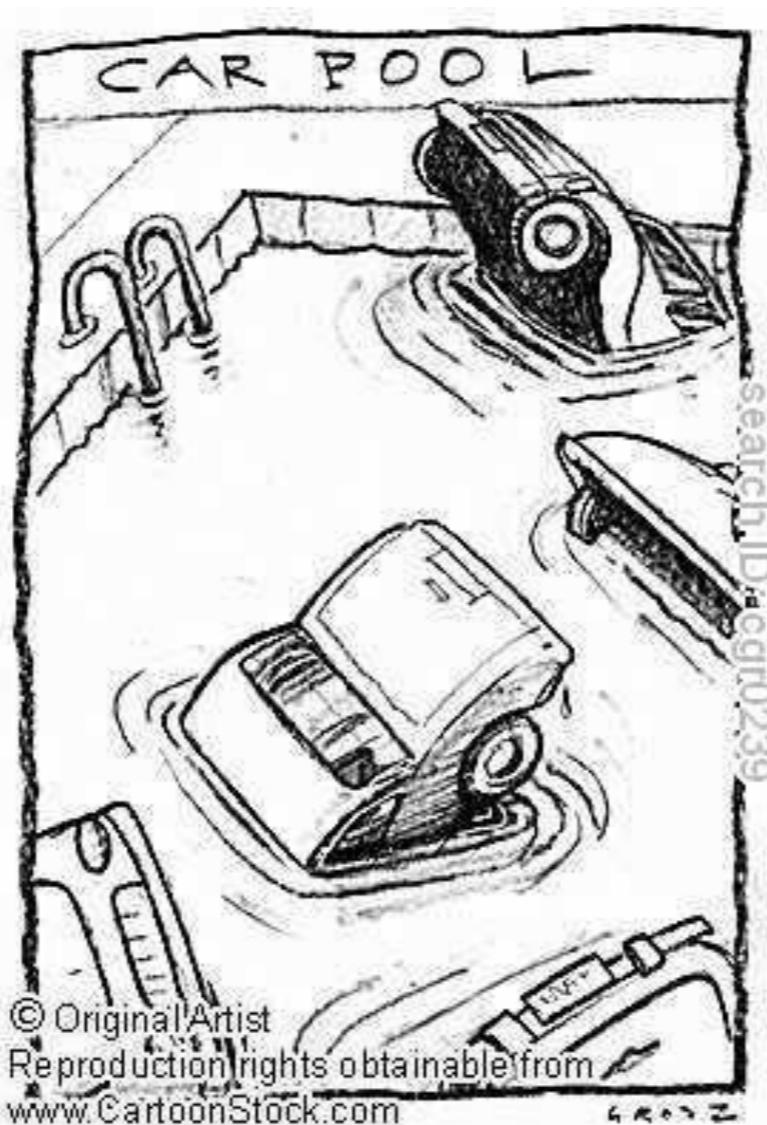
You require a thread to have a particular priority. Changing priority is not recommended for thread's in Thread pool.

# When To Create Thread

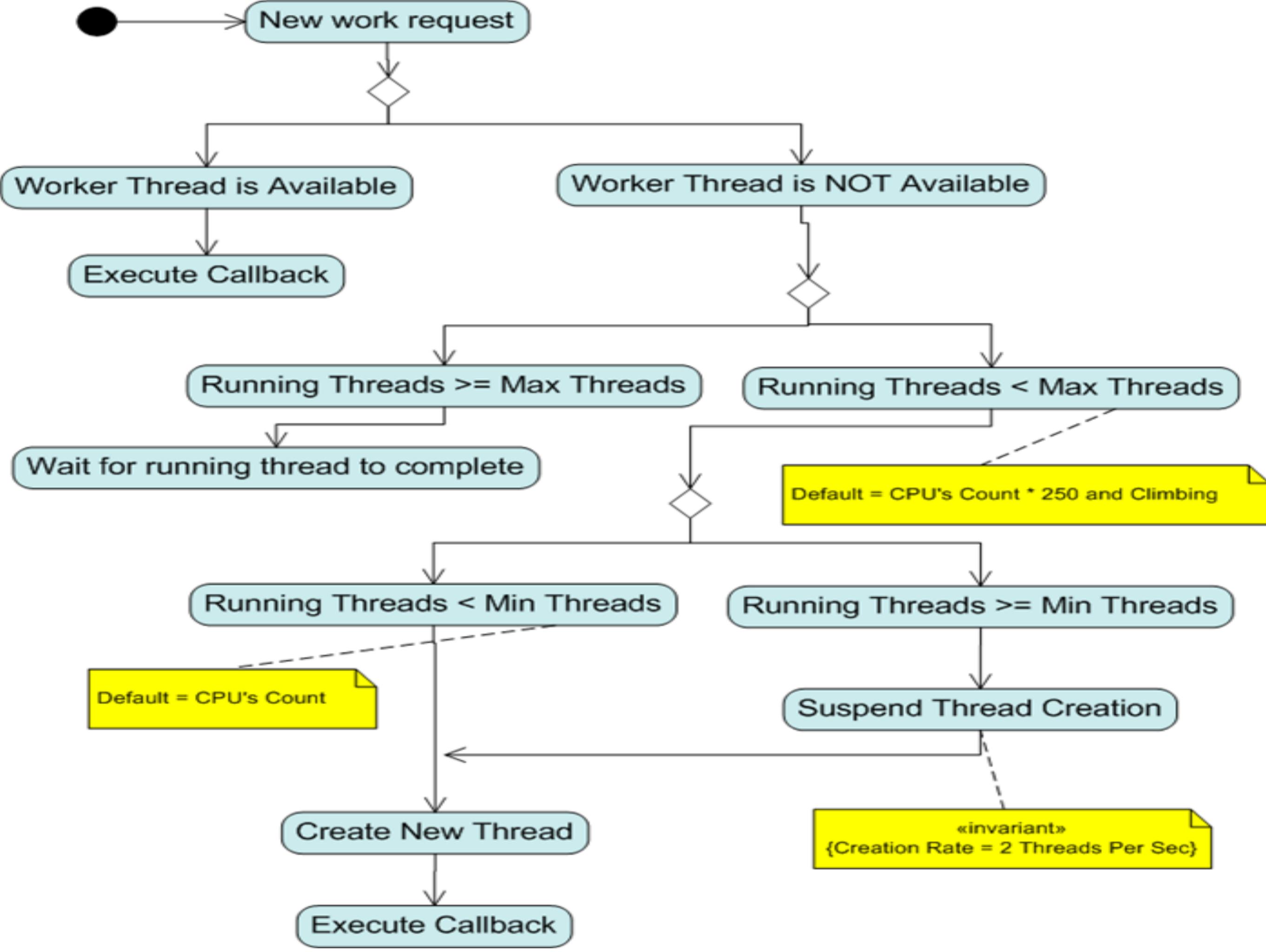


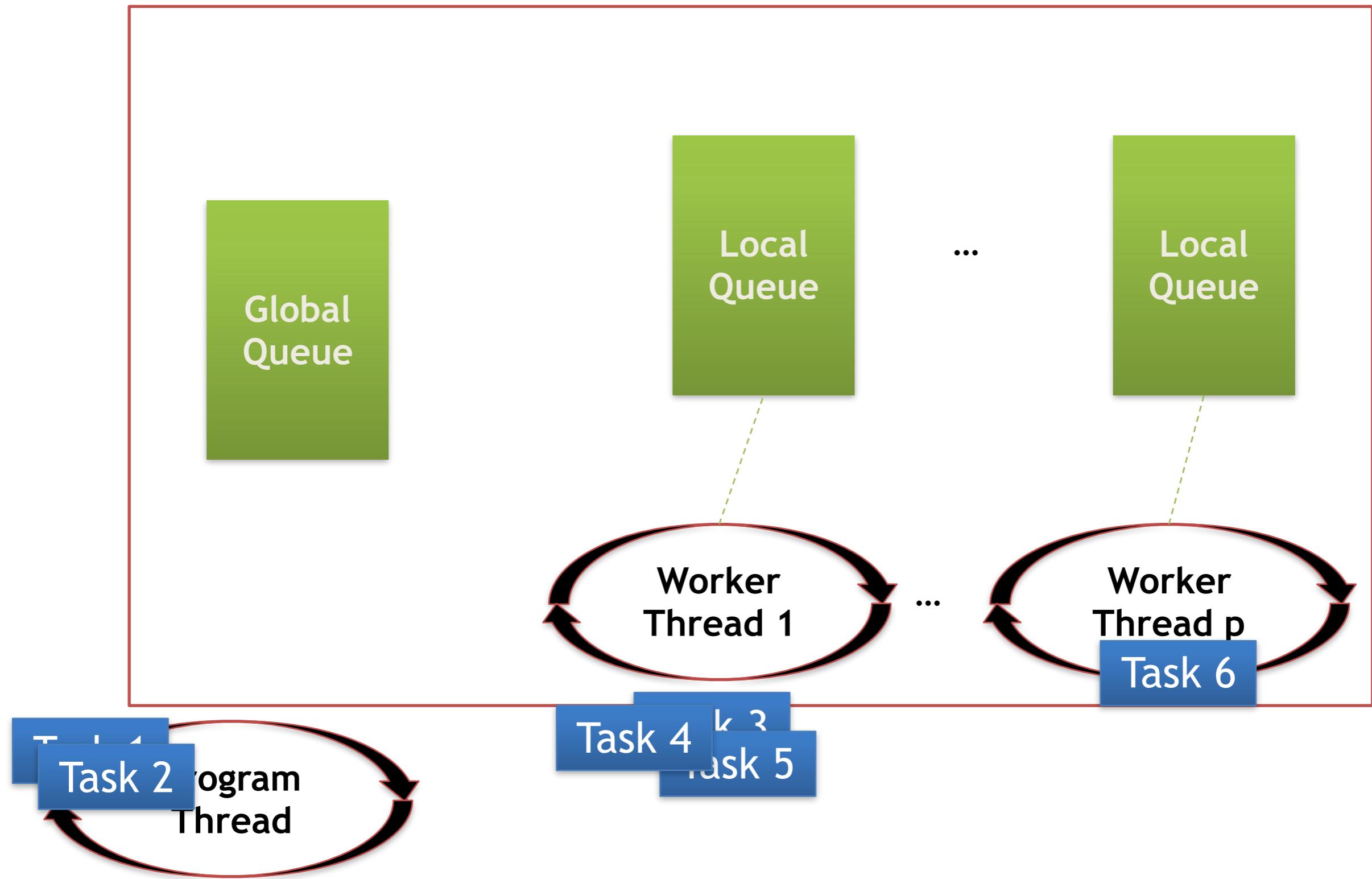
You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.

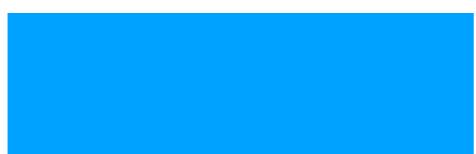
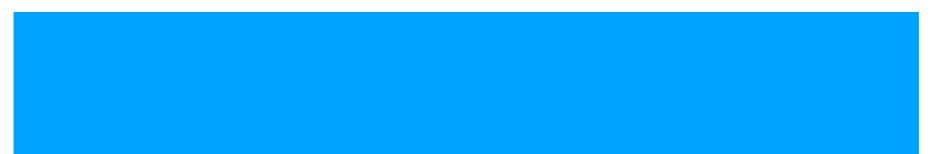
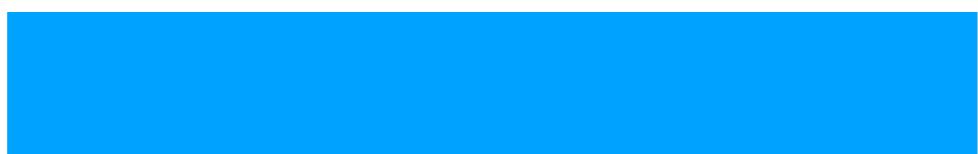
# When To Create Thread



You need to have a stable identity associated with the thread, or to dedicate a thread to a task.







# 3 Approaches to Parallelism

## 1. Imperative data parallelism

- Applicable to common data oriented operations such as for and foreach loops.

## 2. Imperative task parallelism

- For performing parallel tasks via expressions and statements.

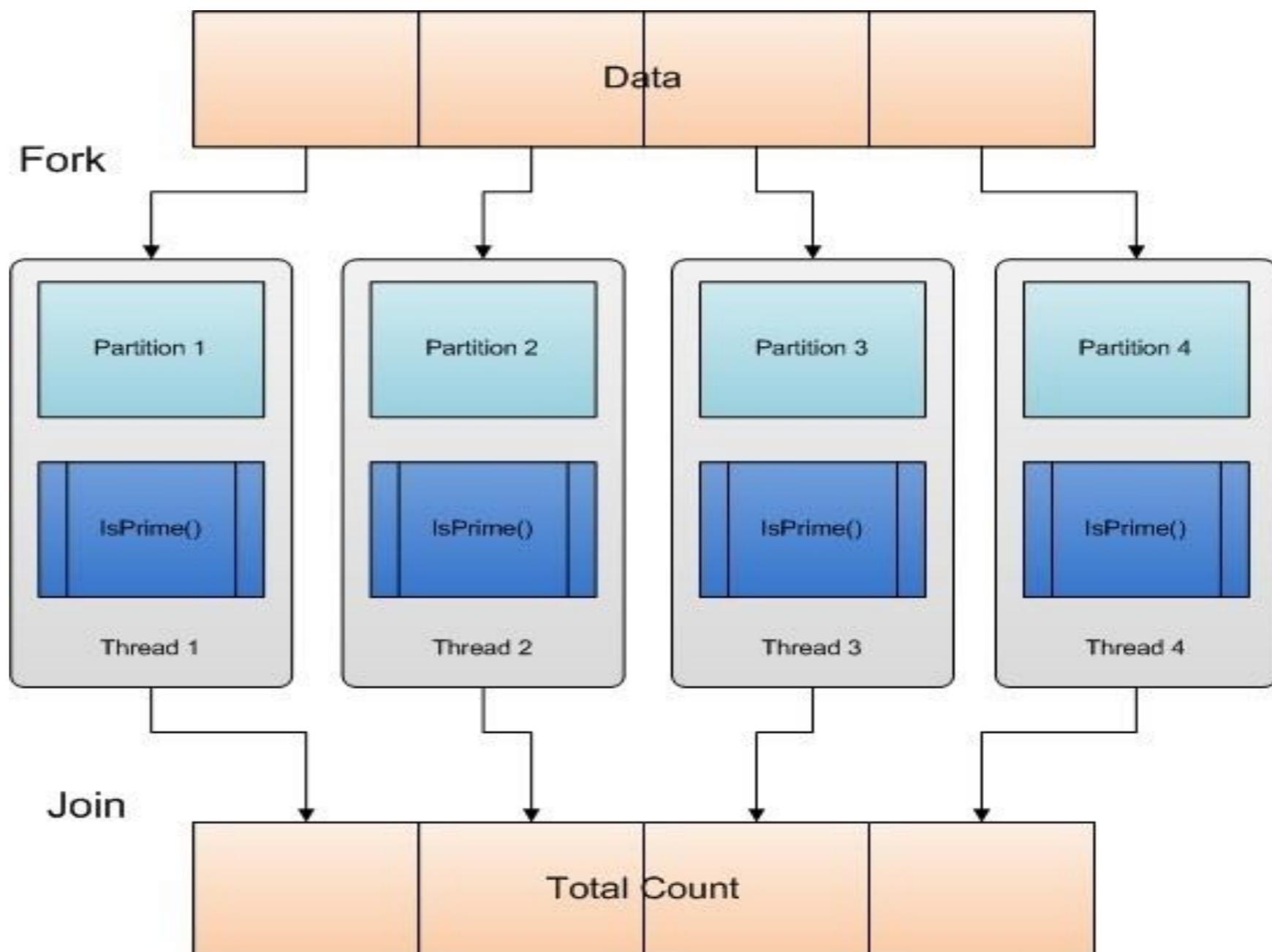
## 3. Declarative data parallelism

- Applicable while querying data.

job1(){ logic 1 }  
job2(){ logic 2 }  
job3(){ logic 3 }

data1, data2, data2  
job( data ){ logic }

	<b>Partitions work</b>	<b>Collates results</b>
<b>Parallel Stream</b>	<b>Yes</b>	<b>Yes</b>
Thread class	No	No



# System.Threading

## Parallel.For

```
//for (int i = 0; i < 100; i++)
System.Threading.Parallel.For(0, 100,
    delegate(int i){ // parallel work here } );
```

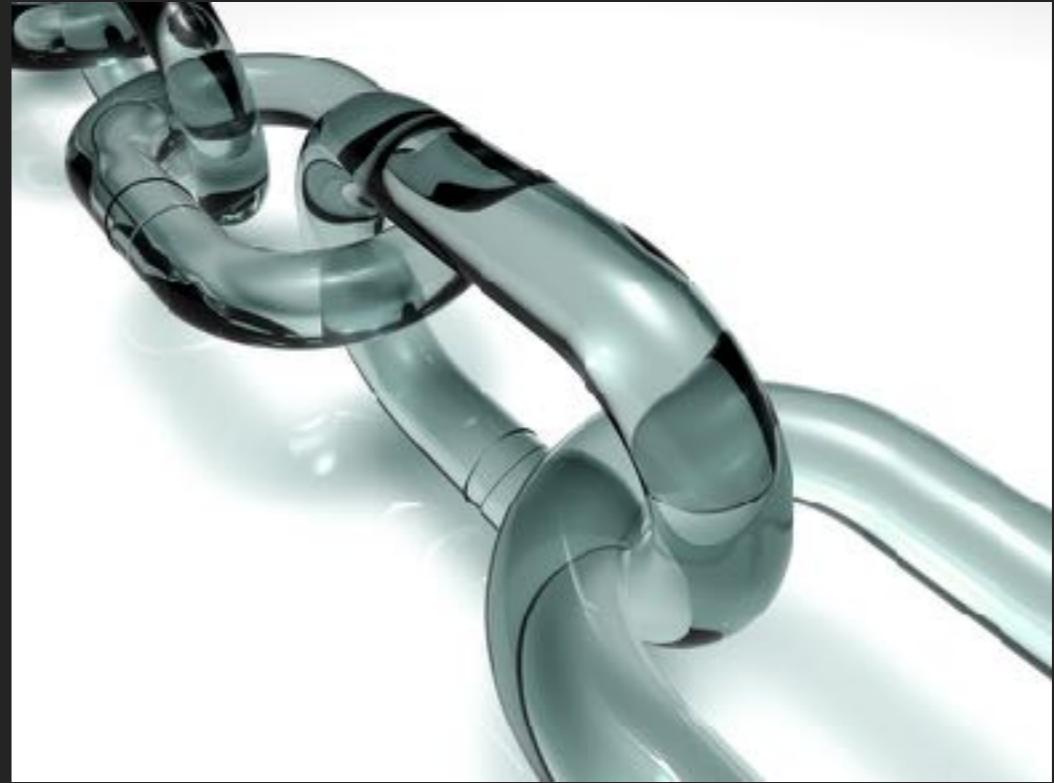
## Parallel.ForEach

```
List<int> list = new List<int>();
Parallel.ForEach(list,
    item =>{ // parallel work here } );
```

## Parallel.Invoke

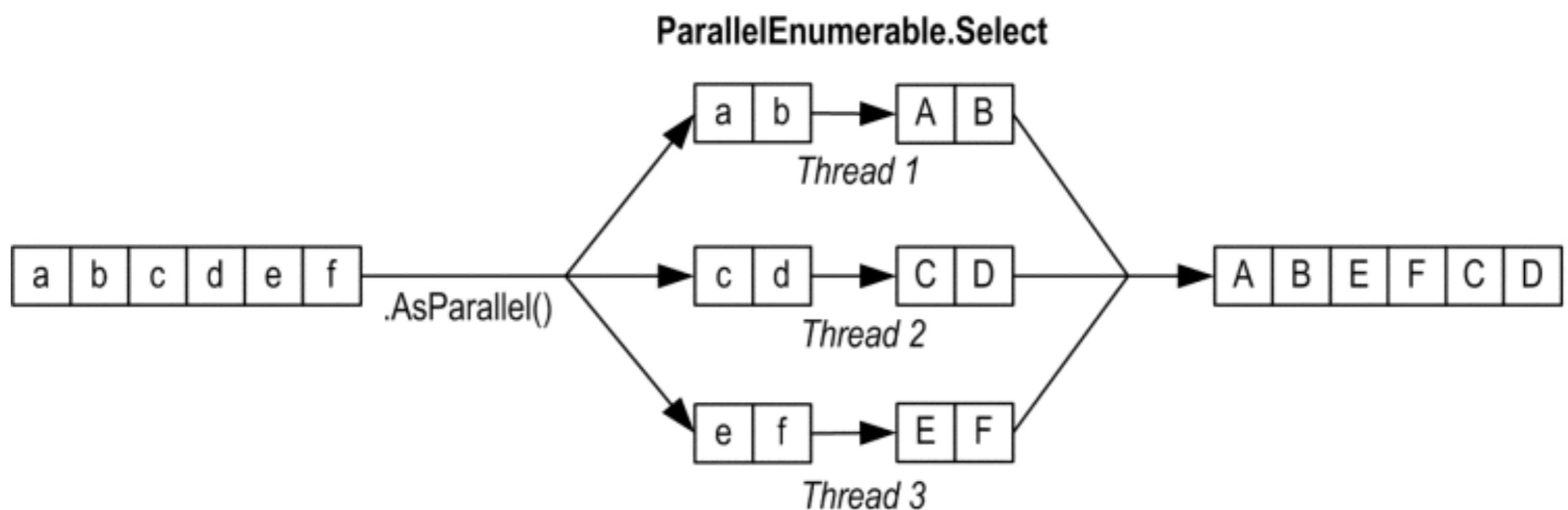
```
Parallel.Invoke(A, B, C, D, E);
public static void A(){}
public static void B(){}
```

# System.Threading



## PLinq

```
var q = from p in people.AsParallel()
        where p.Name == queryInfo.Name &&
              p.State == queryInfo.State &&
              p.Year >= yearStart &&
              p.Year <= yearEnd
        orderby p.Year ascending
        select p;
```

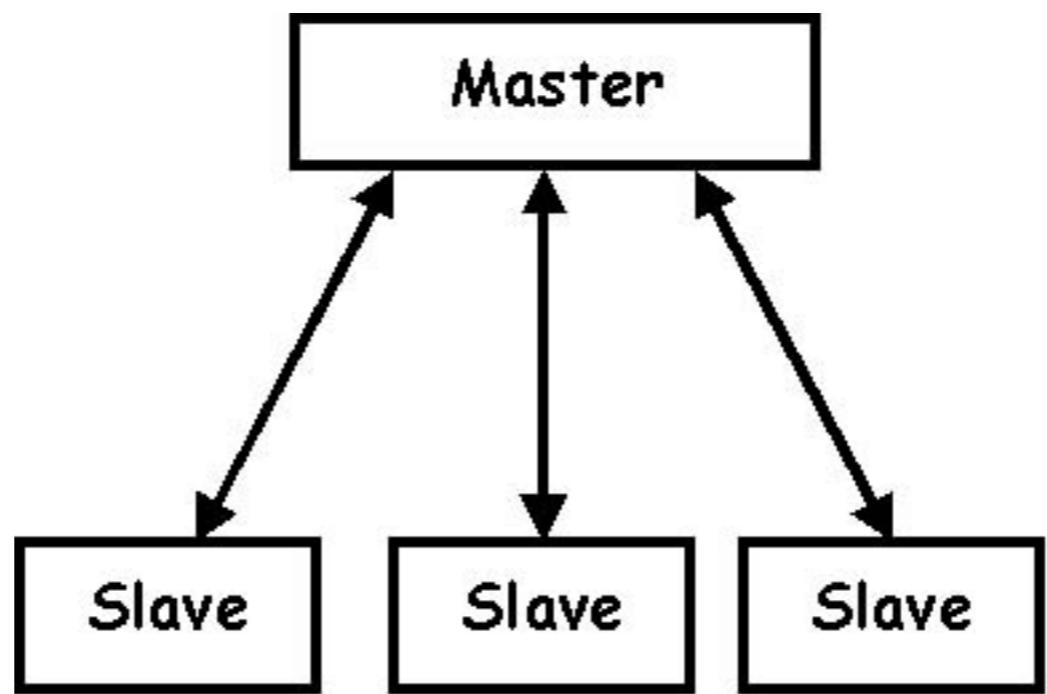


```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ToArray()
```

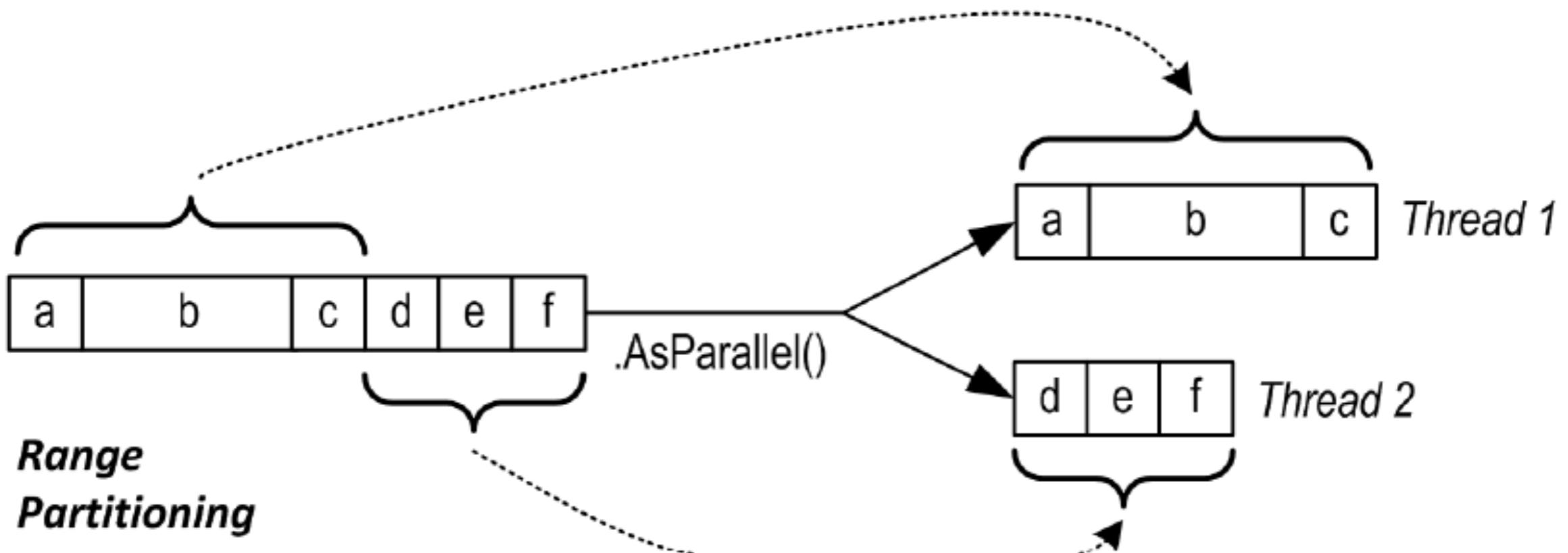
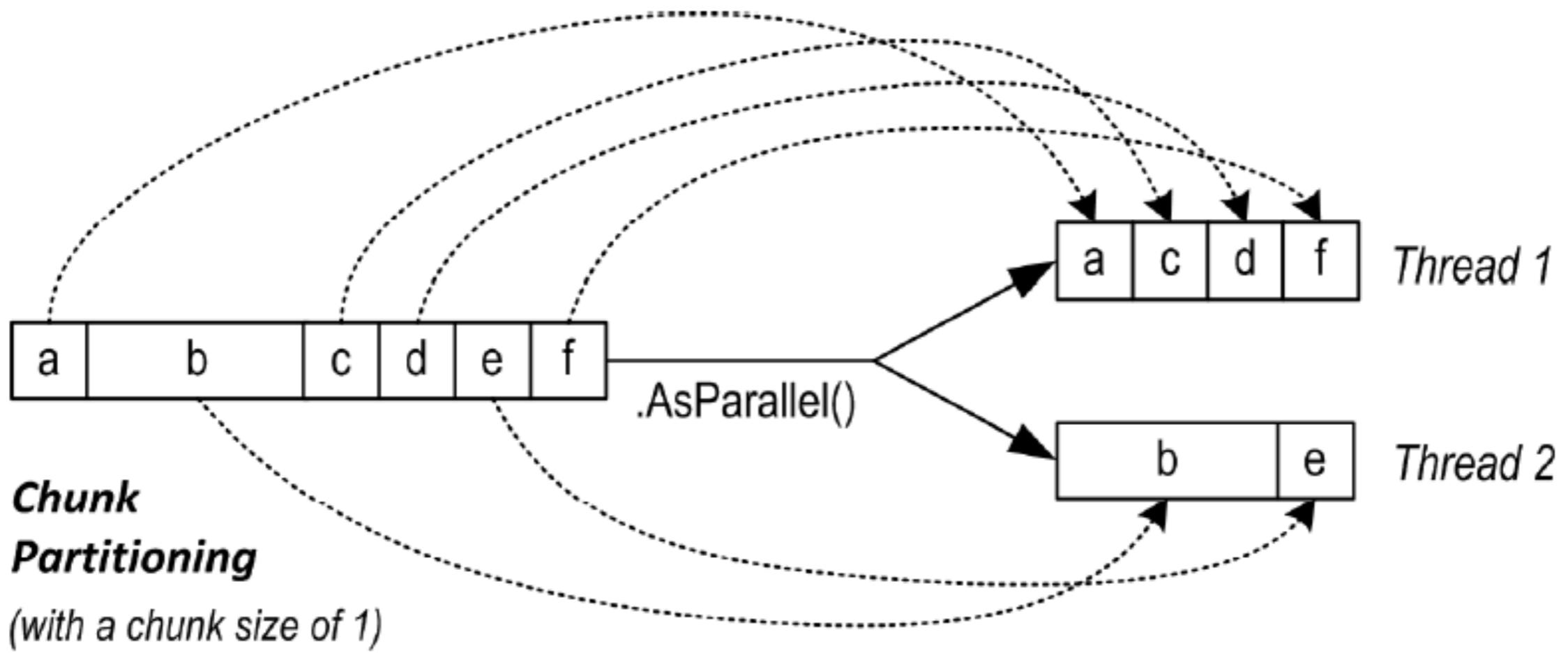
```
employees.parallelStream().forEach(Main::save);
```

v/s

```
ExecutorService executor =  
Executors.newFixedThreadPool(100);  
for(Employee e: employees)  
{  
    executor.execute(()-> save(e));  
}
```



**Why Isn't `.AsParallel()` the Default?**



# Partitioning Algorithms

- Several partitioning schemes built-in

- **Chunk**

- Works with any `IEnumerable<T>`
    - Single enumerator shared; chunks handed out on-demand



- **Range**

- Works only with `IList<T>`
    - Input divided into contiguous regions, one per partition



- **Stripe**

- Works only with `IList<T>`
    - Elements handed out round-robin to each partition



- **Hash**

- Works with any `IEnumerable<T>`
    - Elements assigned to partition based on hash code



- Custom partitioning available through `Partitioner<T>`

- `Partitioner.Create` available for tighter control over built-in partitioning schemes

Partitioning Strategy	Partitioning Strategy	Relative performance
<i>Chunk</i>	<i>Dynamic</i>	<i>Average</i>
Range	Static	Poor to excellent
<i>Hash</i>	<i>Static</i>	<i>Poor</i>
Stripe	Static	Poor to excellent
<i>Custom</i>	<i>Dynamic/static</i>	Poor to excellent

# The cost of I/O

L1-cache	3 cycles
L2-cache	14 cycles
RAM	250 cycles
Disk	41 000 000 cycles
Network	240 000 000 cycles

# Worker threads or I/O threads

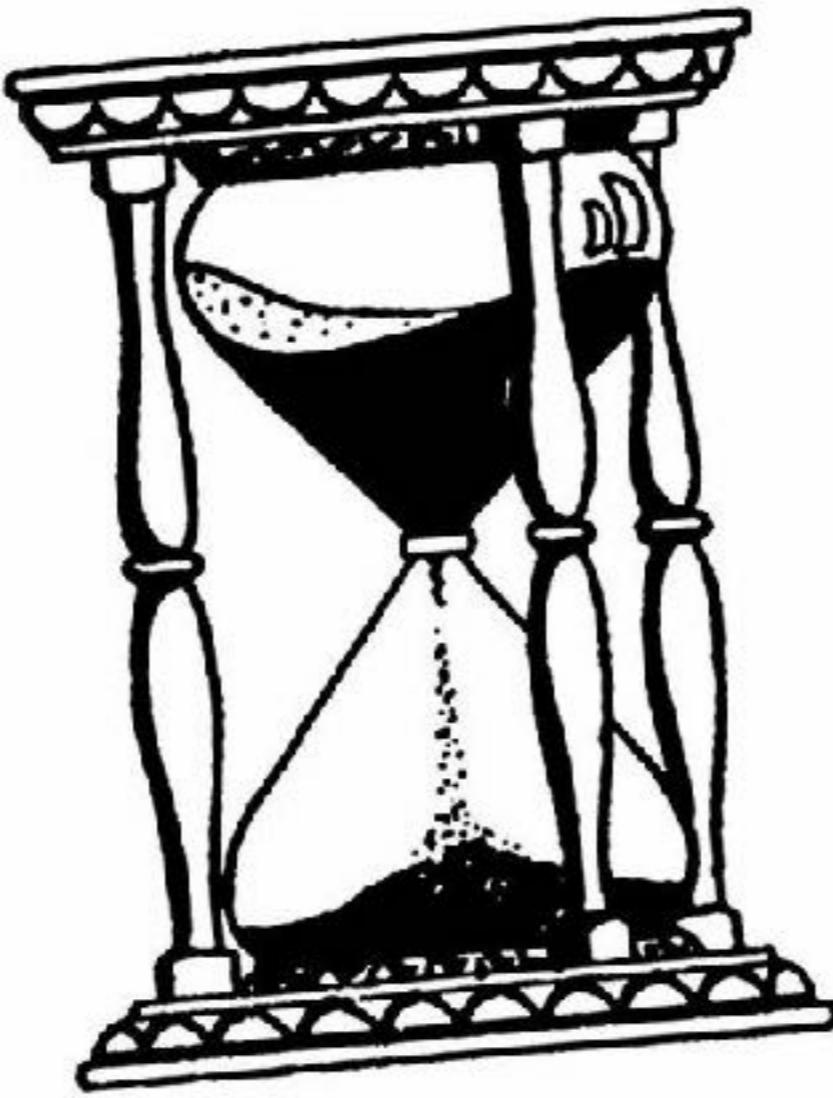


Worker threads (Compute Bound/ CPU Bound)  
I/O threads (Device Bound).



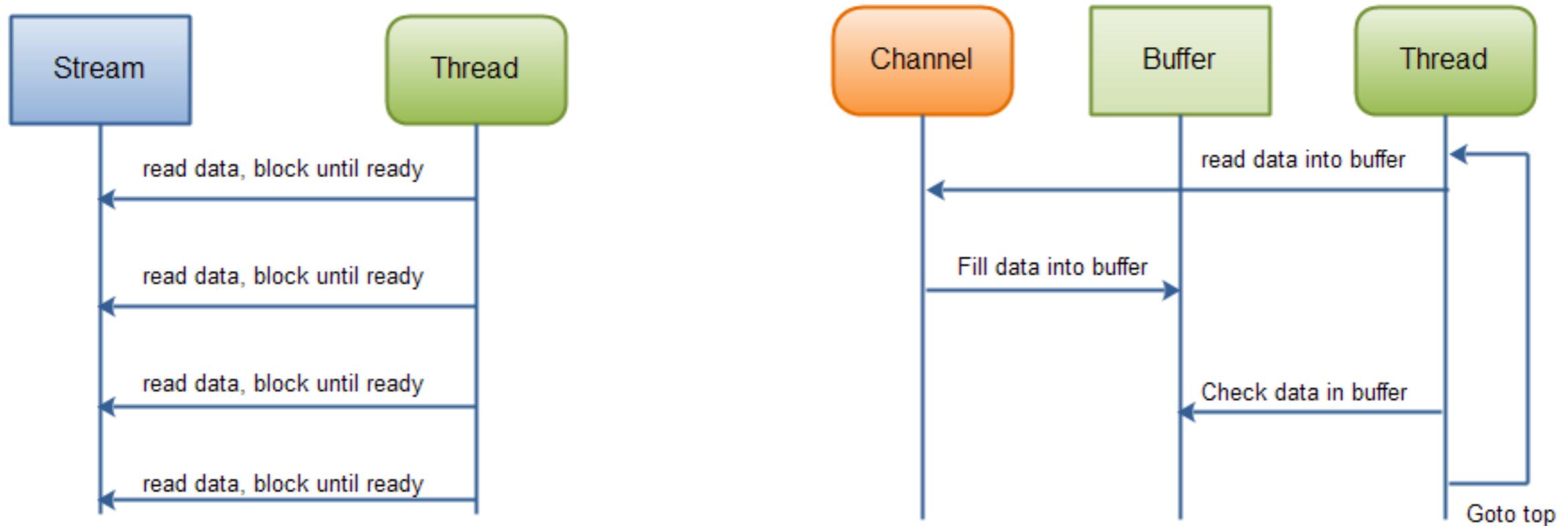
When performing asynchronous I/O-bound operations, you have a device driver doing the work for you and no threads are required.

- Profiler
  - CPU
  - Memory
  - Thread



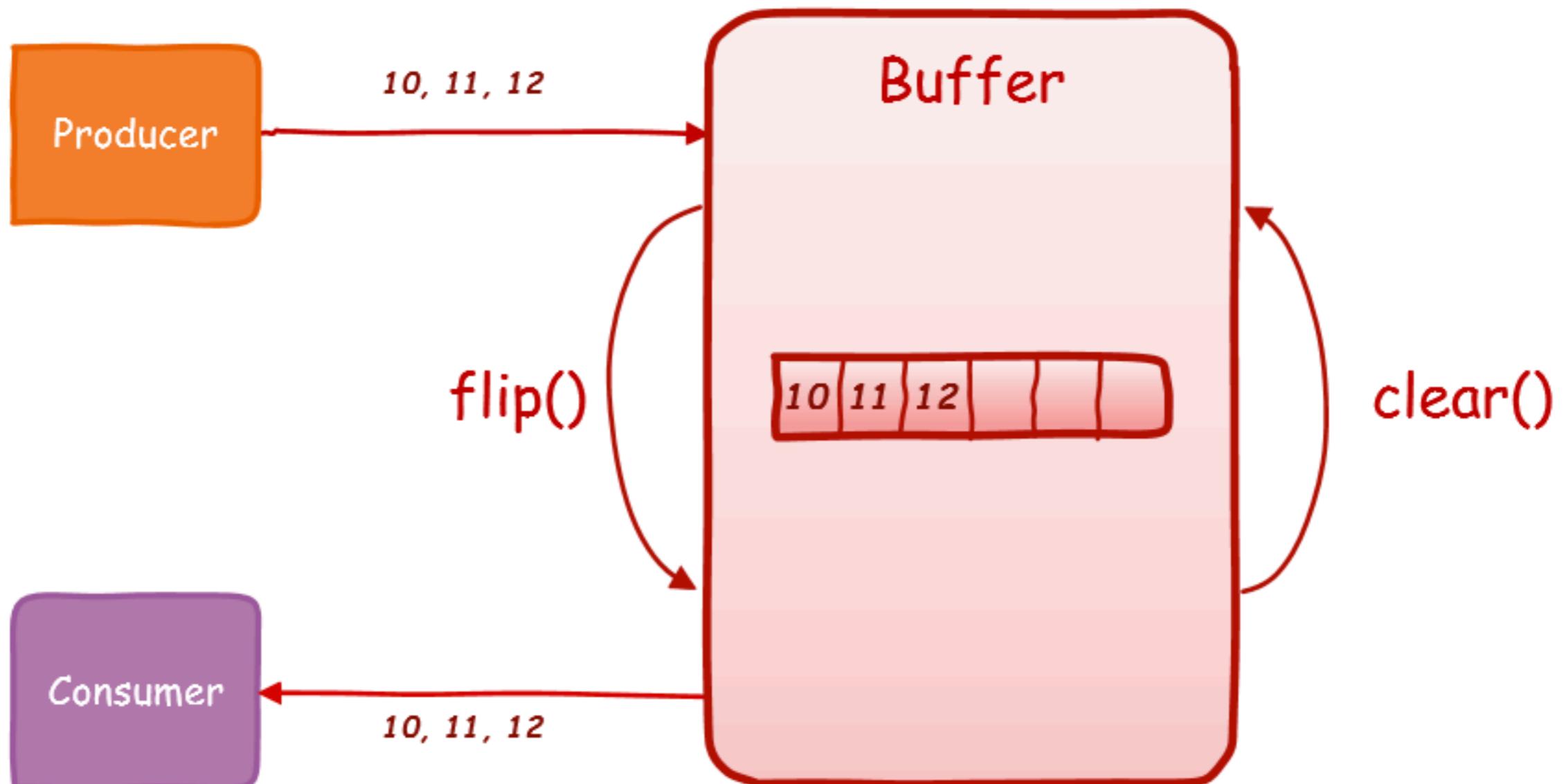
All synchronous I/O operations suspend the calling thread while the underlying hardware (disk-drive, network interface card, etc.) performs the work.

# Blocking vs. Non-blocking IO



## Stream Oriented vs. Buffer Oriented

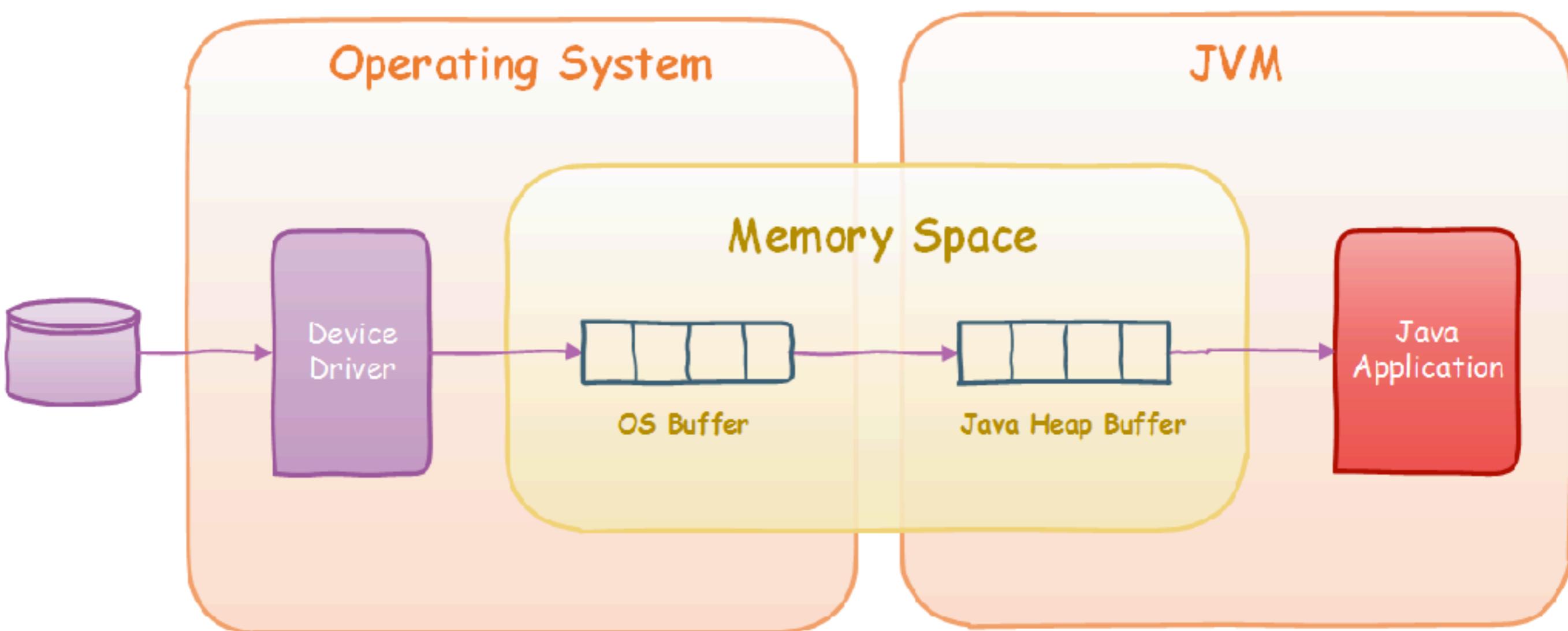
# Filling Mode



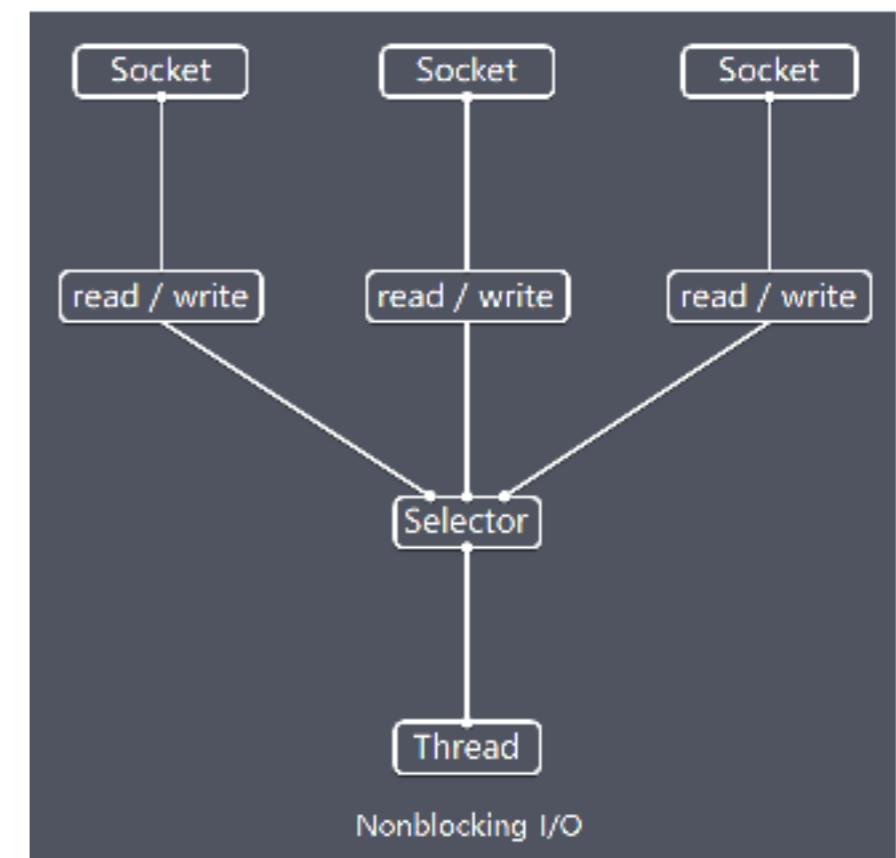
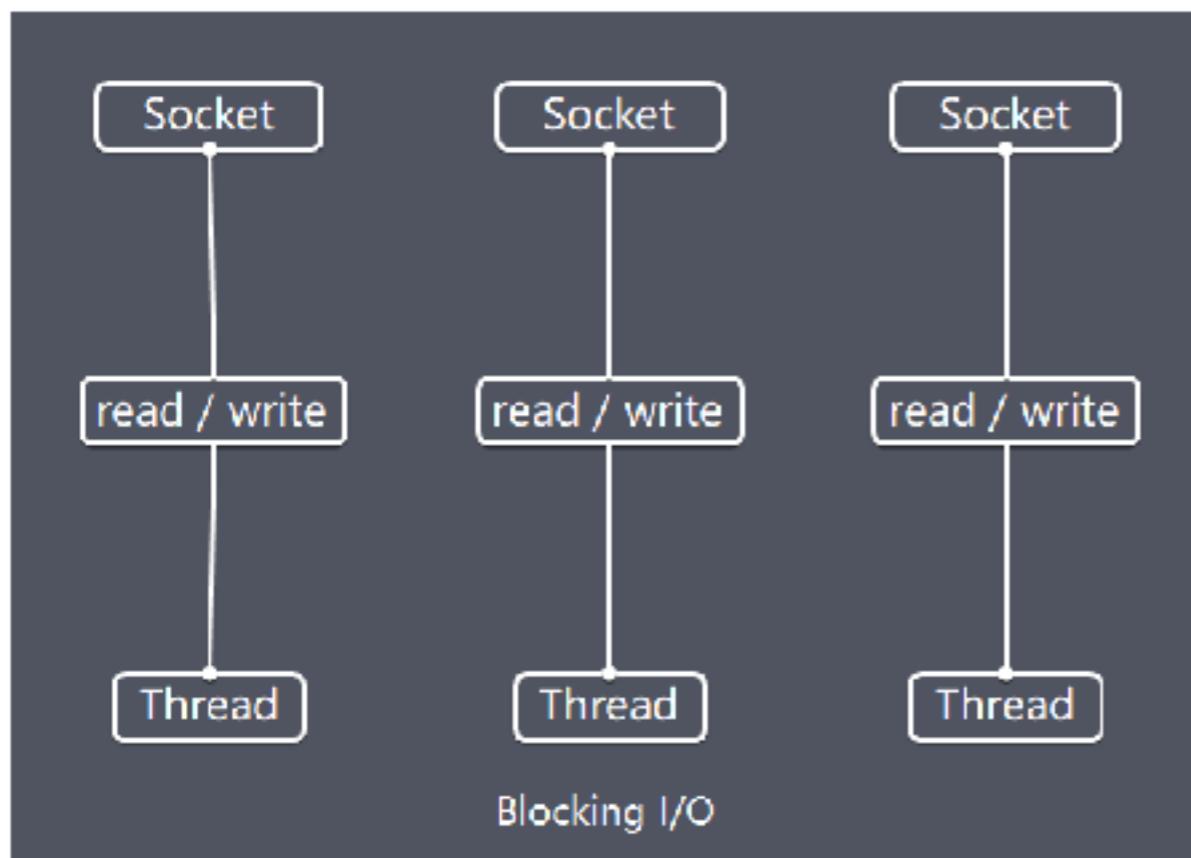
# Draining Mode

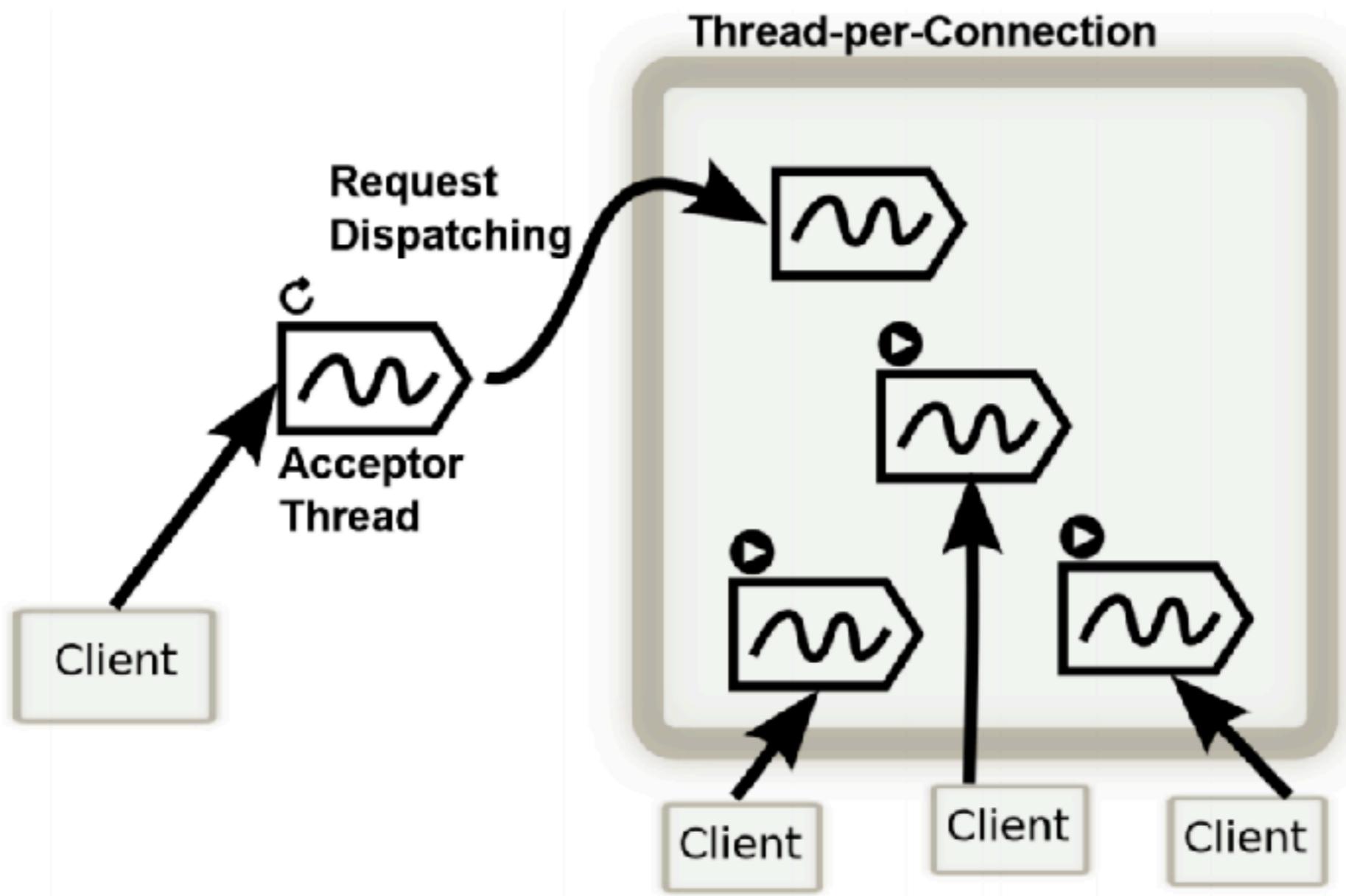
The buffer is in **filling mode**. After the producer has finished writing data, the buffer is then **flipped** to prepare it for **draining mode**. At this point, the buffer is ready for the consumer to read the data. Once done, the buffer is then **cleared** and ready for writing again.

# Buffers - Direct and Non-Direct

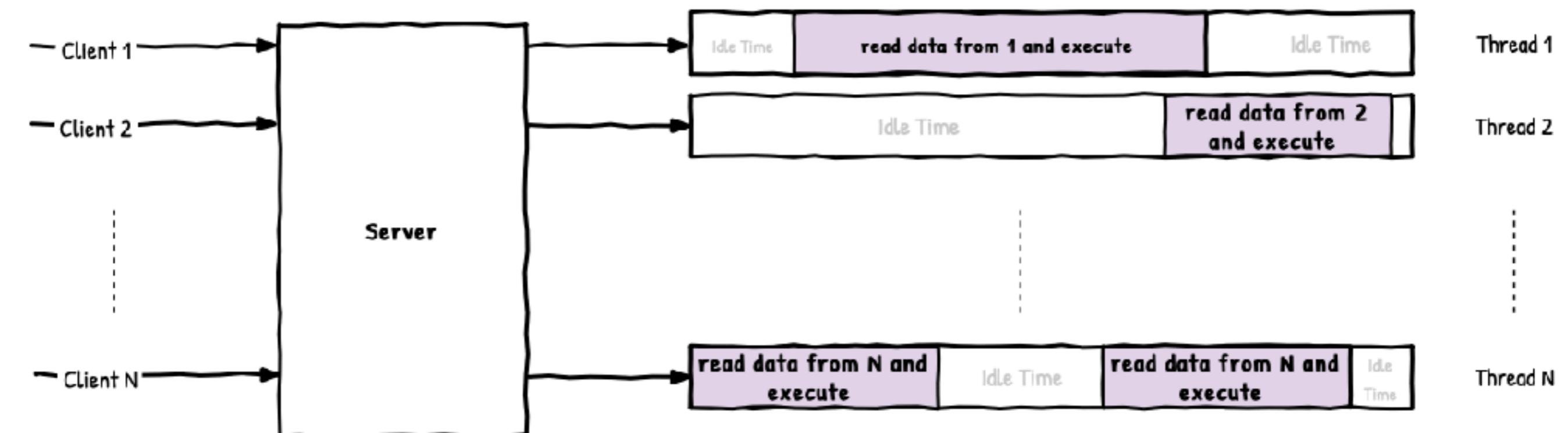


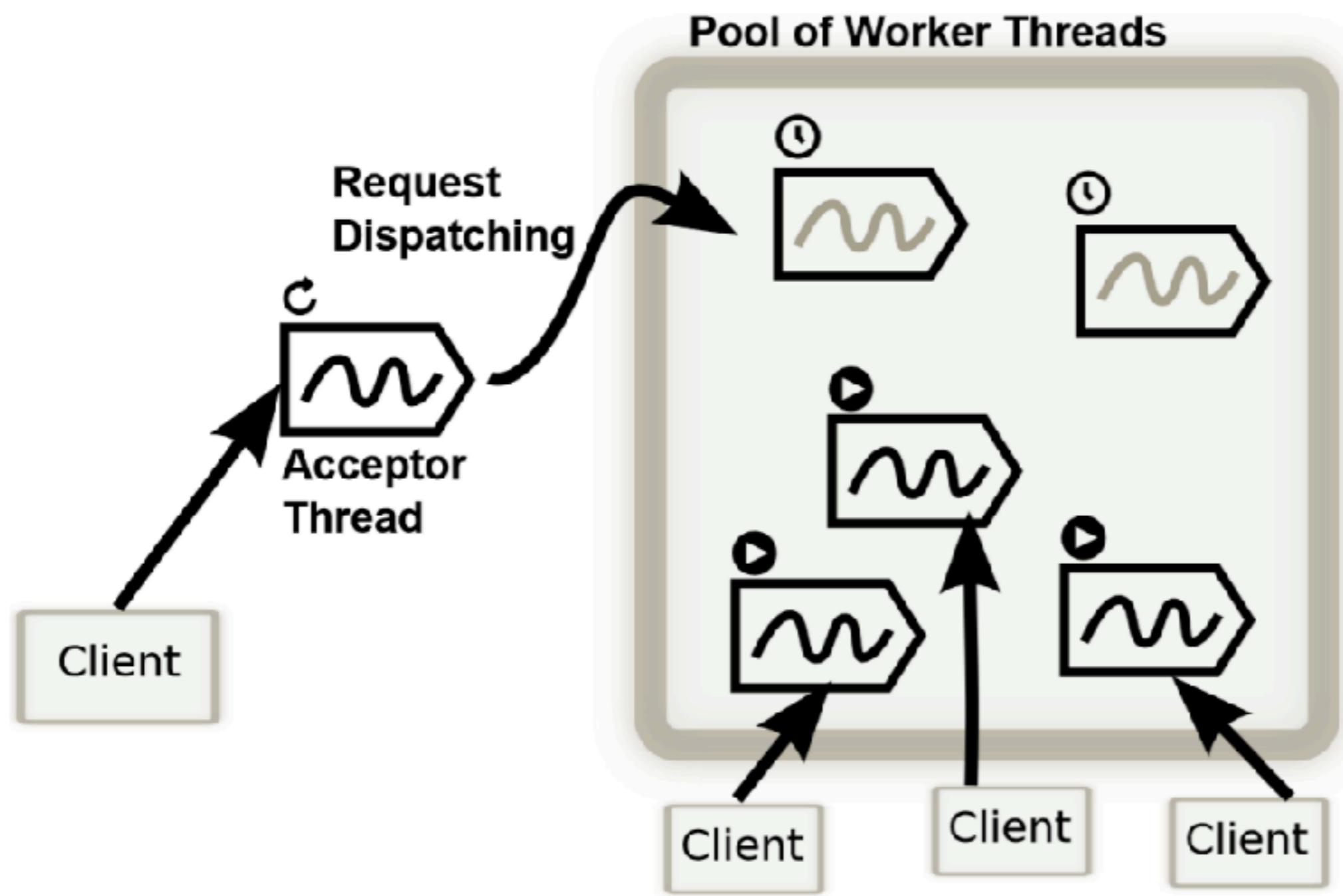
```
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(100);
```

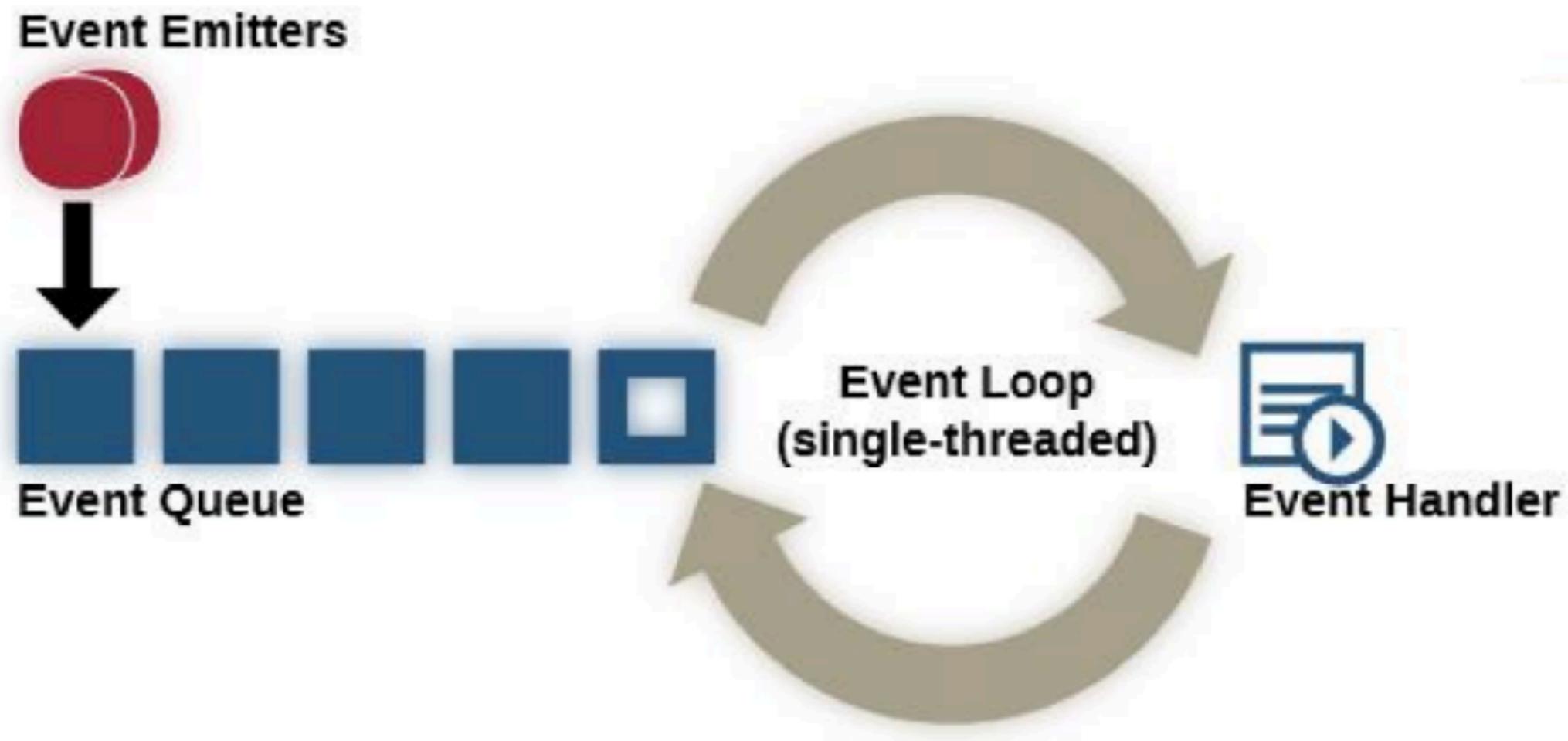




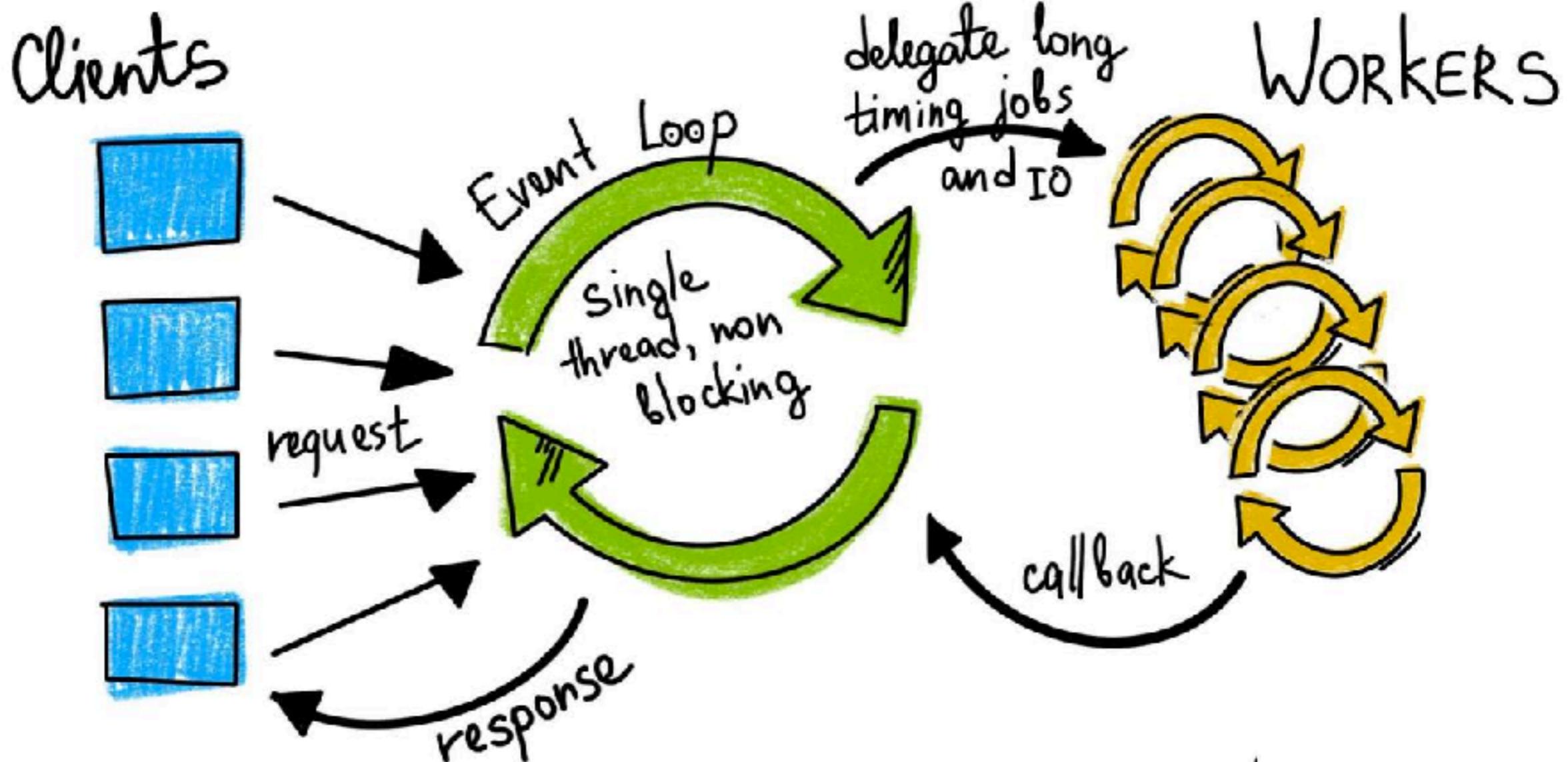
## Server with a thread per connection (Blocking sockets)



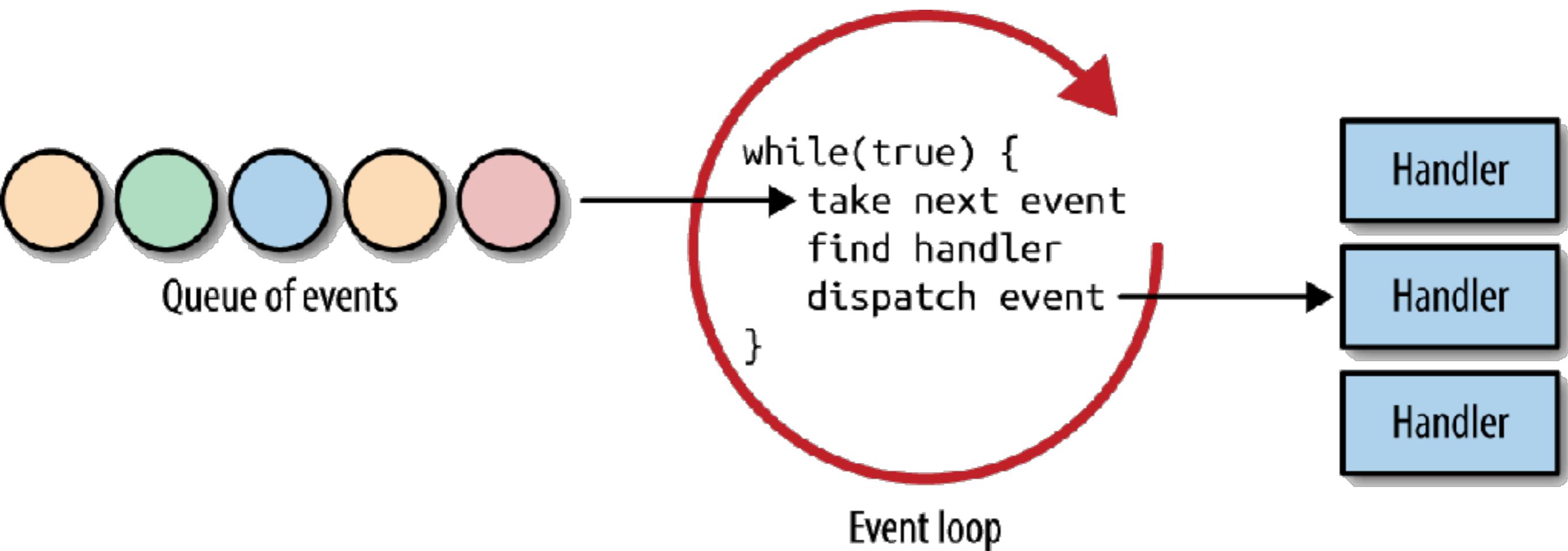


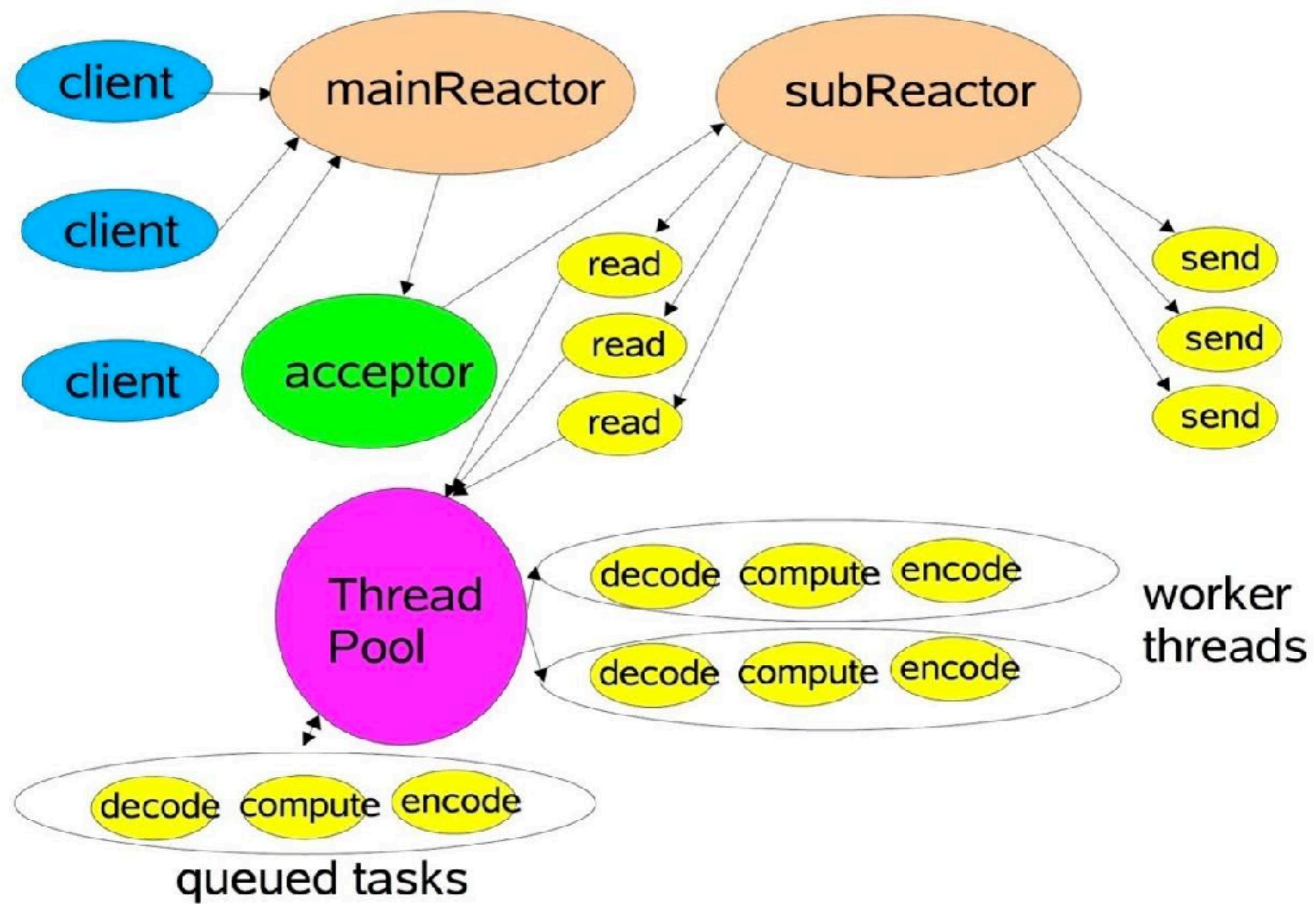


The event demultiplexor delivers the requested events to the event handlers.



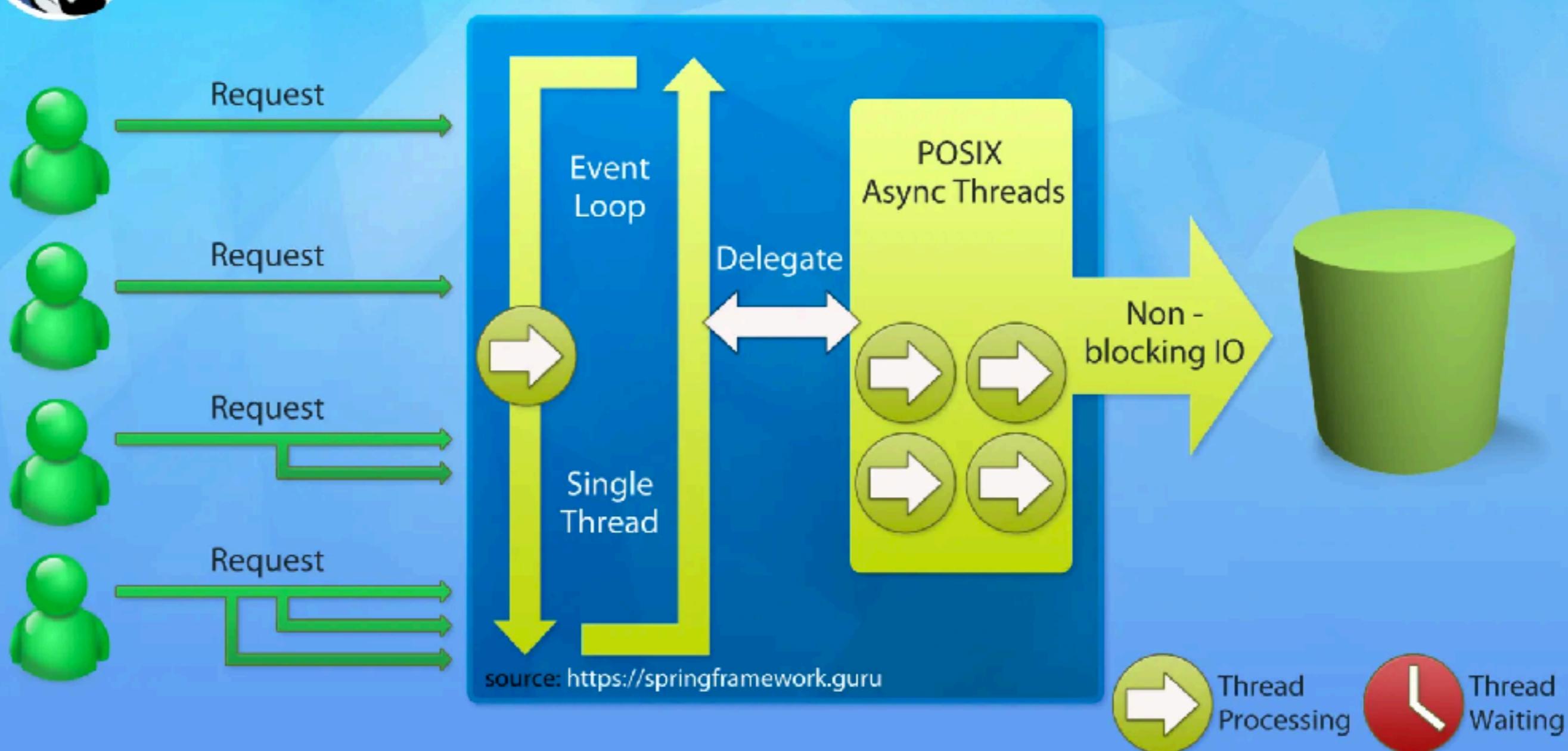
@luminousmen.com



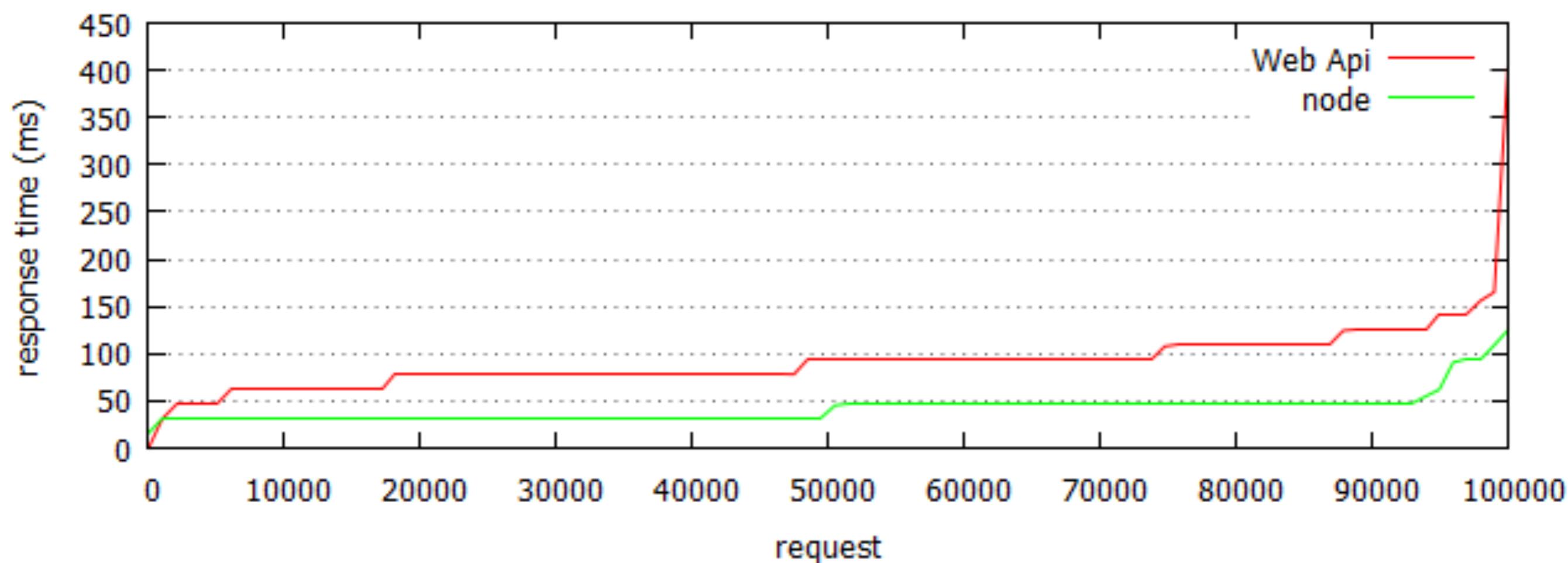


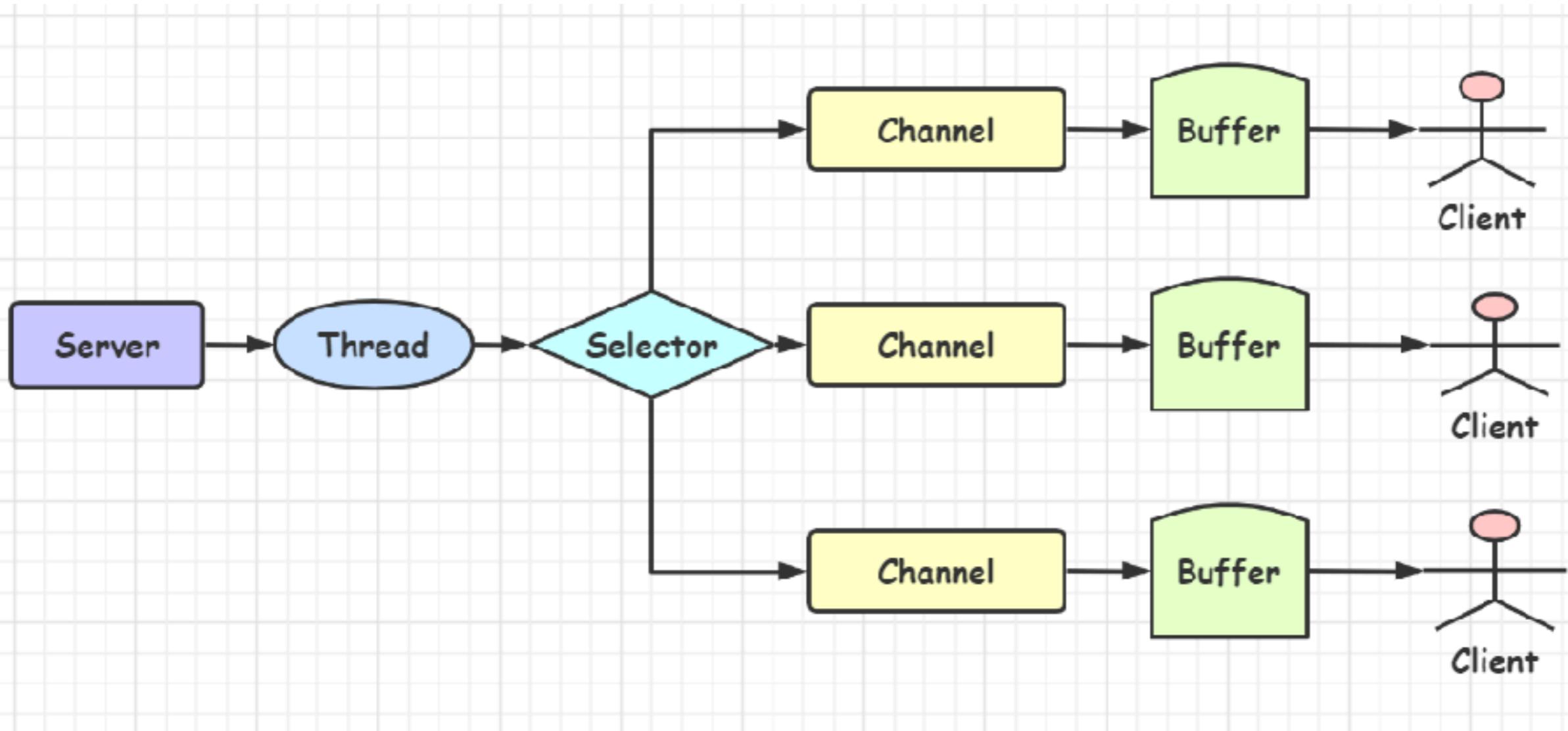


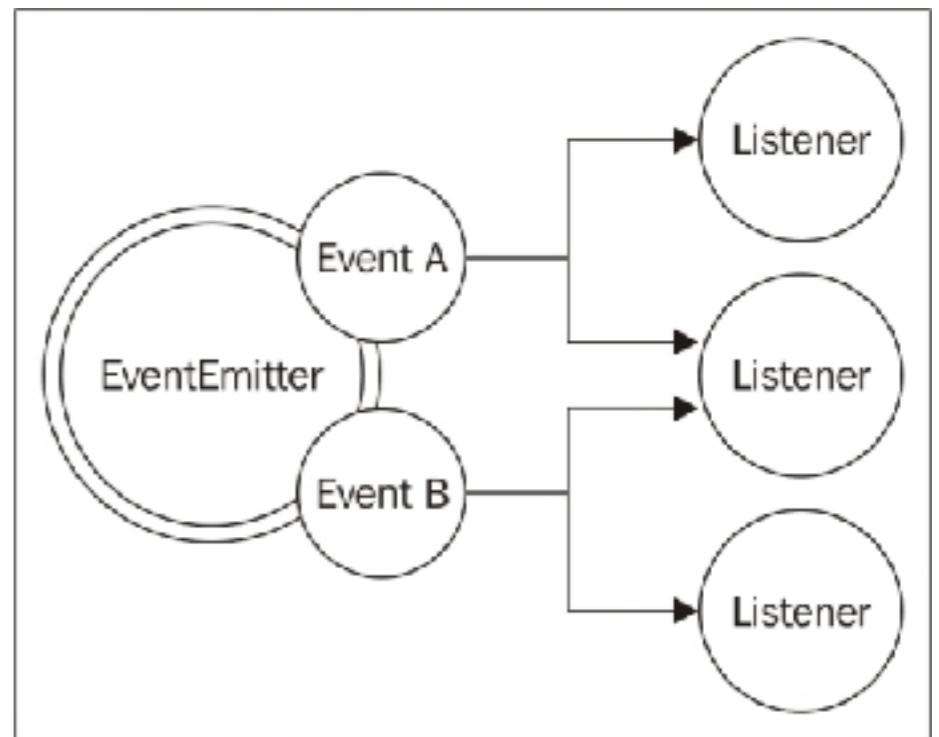
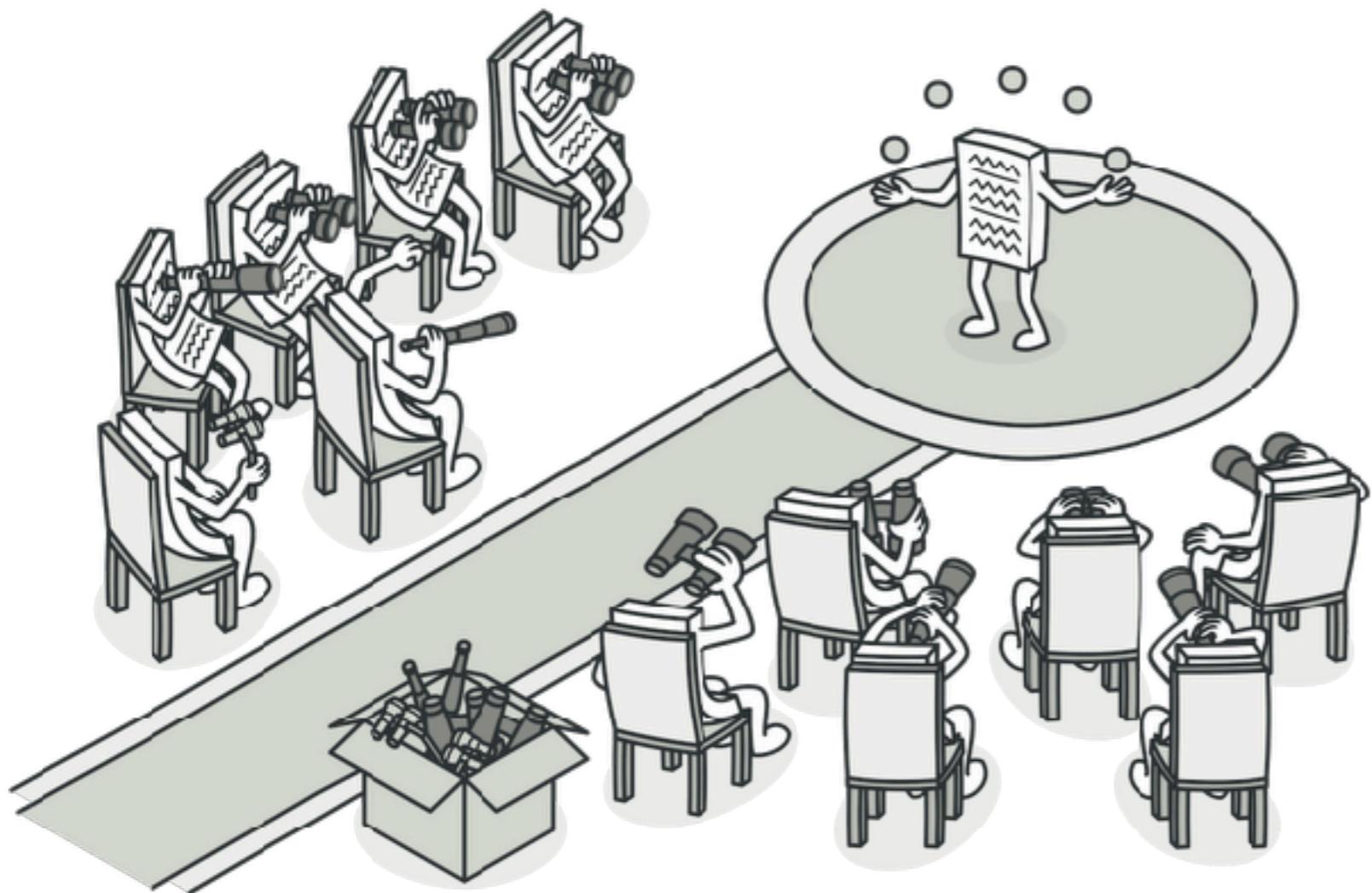
# Node.js Server



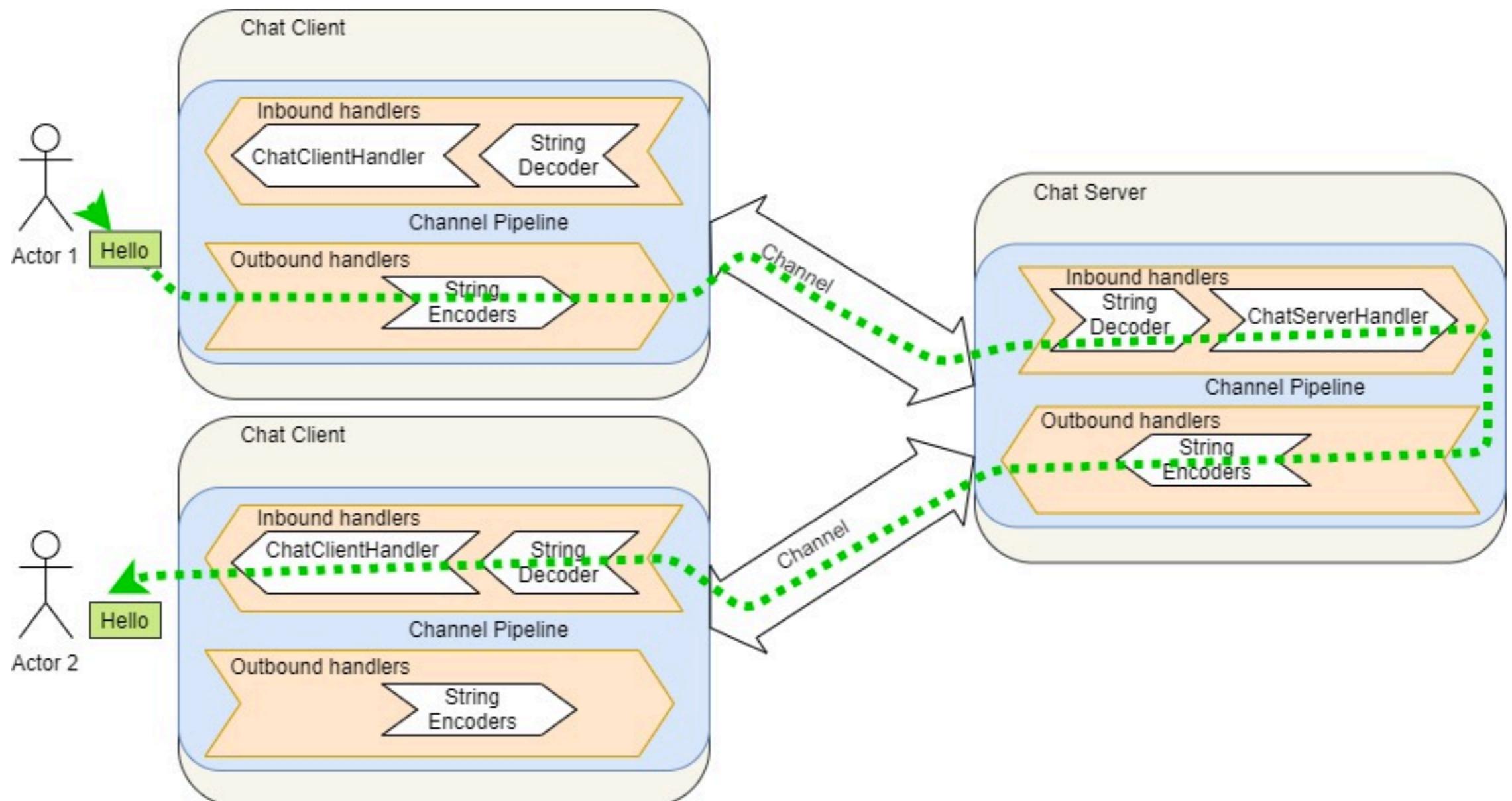
### Web Api vs Node.js





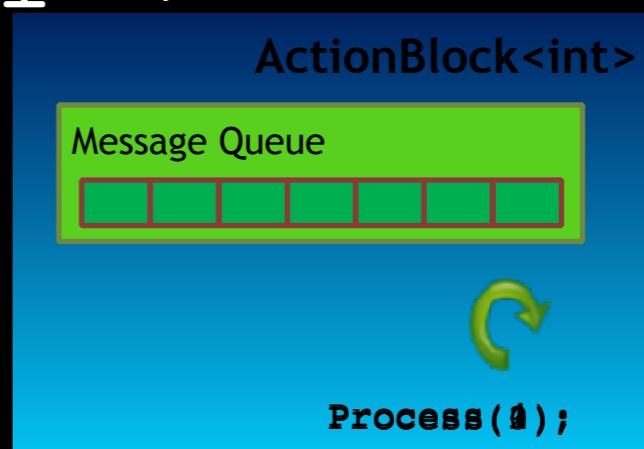


The observer pattern differs from the reactor pattern in a way that each object maintains a list of its dependents. A change in state for an object will simply trigger a notification (i.e., event) to all the dependents signaling a change. The observer pattern implements a one-to-many relationship where multiple dependents could be “observing” the change in an object.

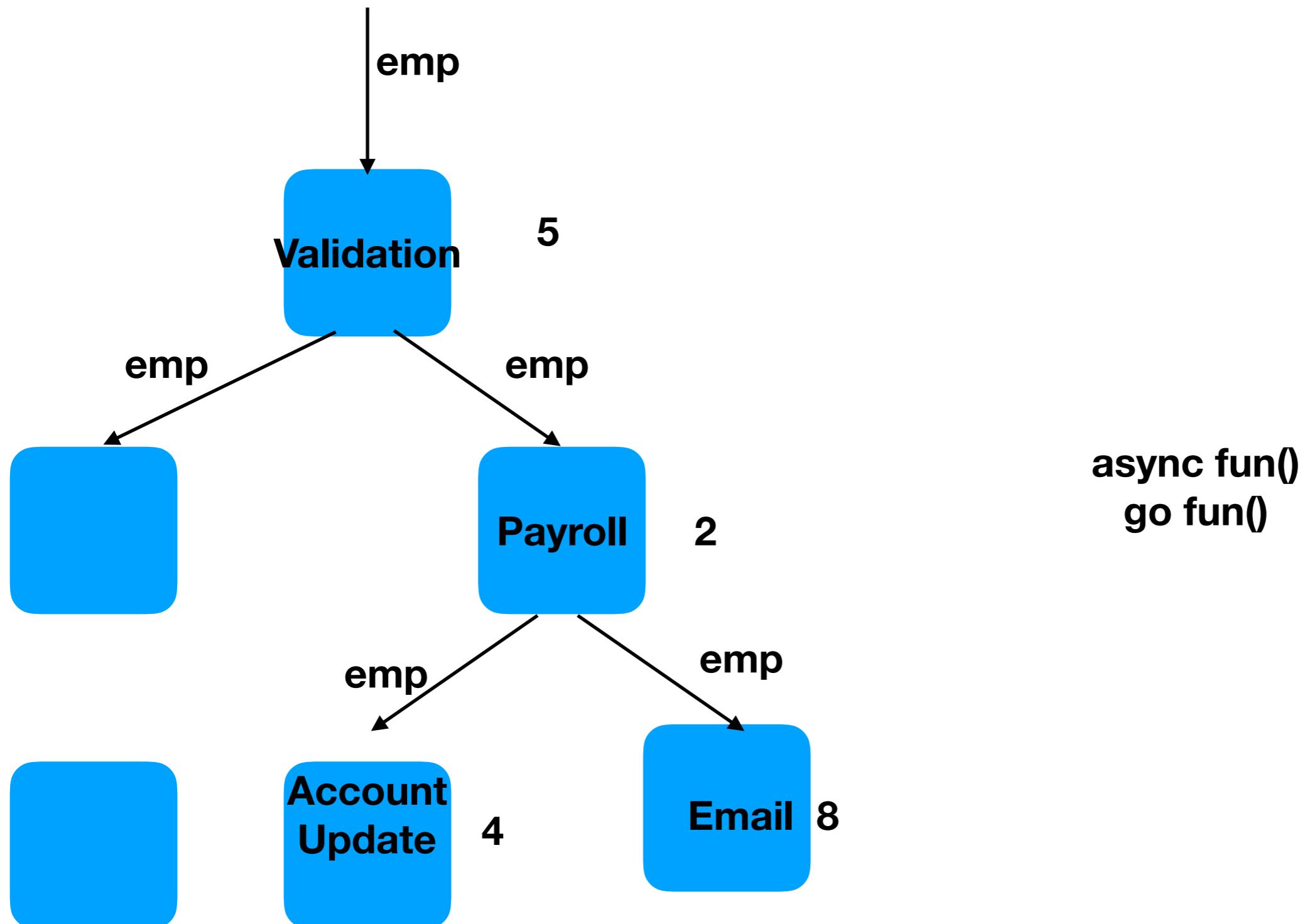


Netty is a low level non-blocking I/O library, used in most reactive solutions including Akka, Vert.x, HBaseAsync, PgAsync.

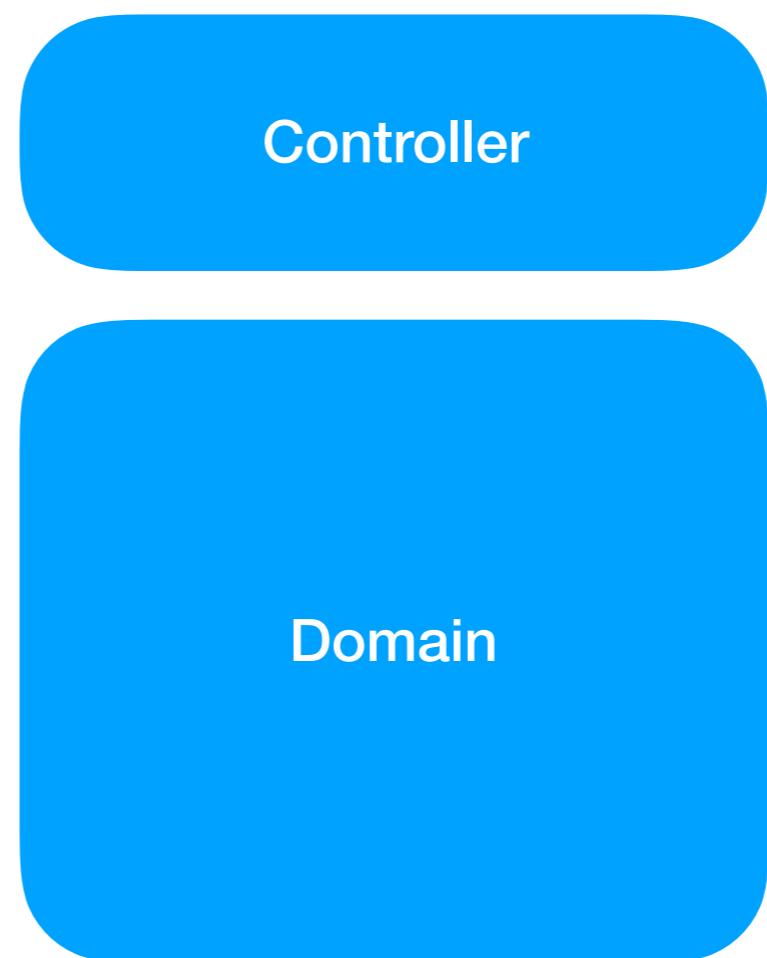
```
var c = new ActionBlock<int>(i =>
{
    Process(i);
}) ;
```

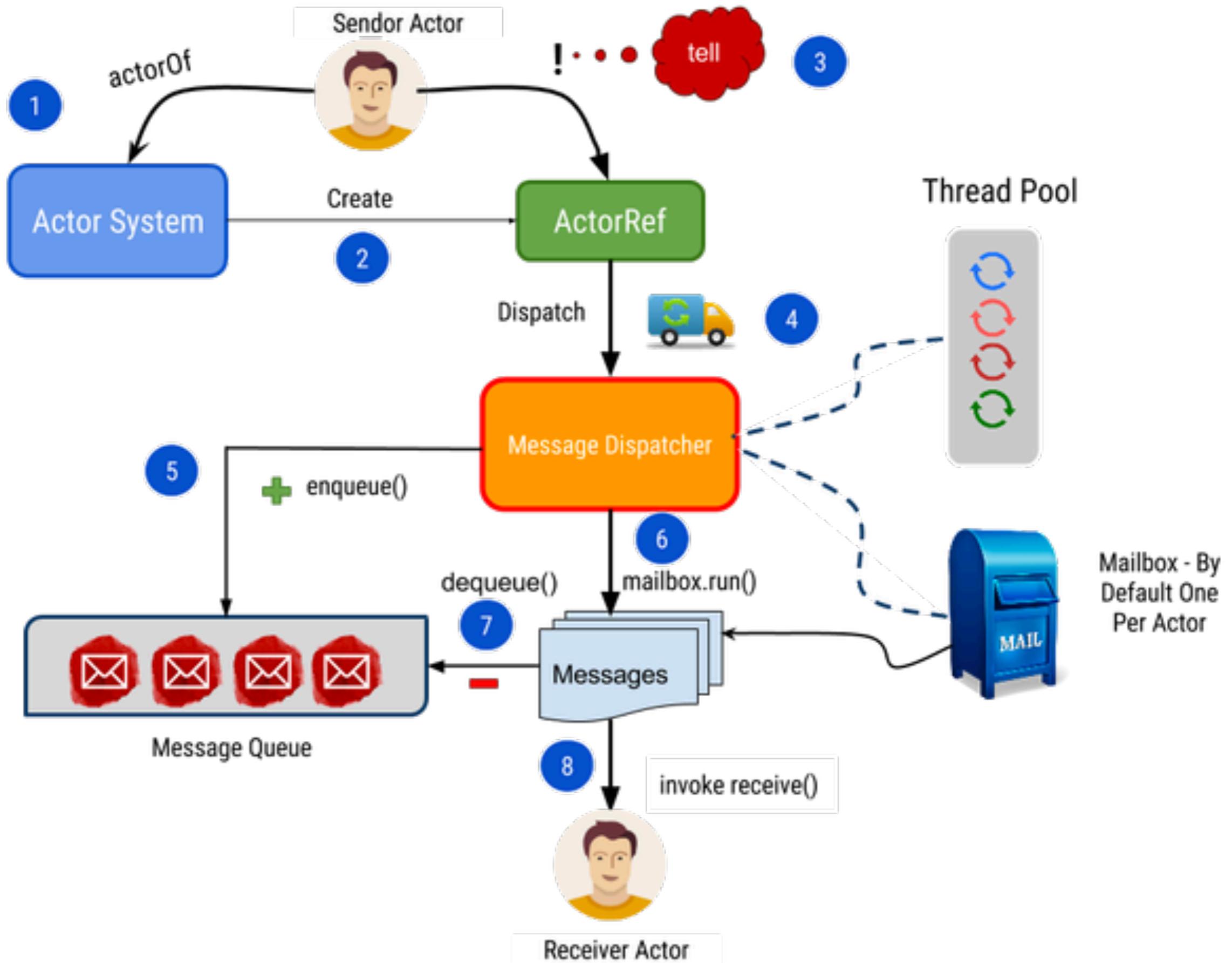


```
for(int i = 0; i < 5; i++)
{
    c.Post(i);
}
```



**Trace log**  
**Distributed trace log (jaeger)**





Netty is actually used within Akka to provide for distributed actors (although Akka is switching over to Spray for this).

# How Windows does Synchronous I/O

.NET

```
FileStream fs = new FileStream(...);  
Int32 bytesRead = fs.Read(...);
```

Win32  
User-Mode

```
ReadFile(...);
```

Windows  
Kernel-Mode

```
(Windows I/O Subsystem Dispatcher code)
```

①

⑨

③

⑧

④

⑦

Your thread blocks here!  
Hardware does I/O;  
No threads involved!

⑥

NTFS Driver

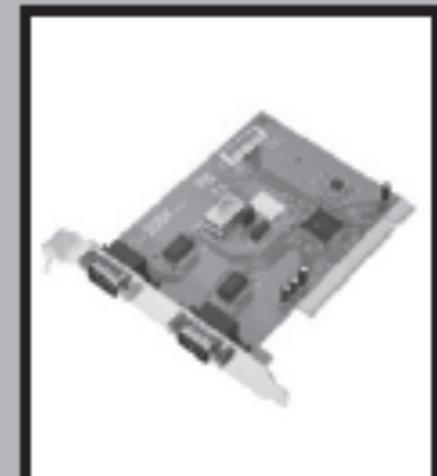
Network



DVD-ROM



RS-232



IRP  
Queue



⑤

## How Windows does Asynchronous I/O

.NET

```
FileStream fs = new FileStream(..., FileMode.Open);
fs.BeginRead(..., AsyncCallback, ...);
```

①

⑦

```
void AsyncCallback(...) { ... }
```

Win32  
User-Mode

```
ReadFile(...);
```

③

IRP

⑥

Windows  
Kernel-  
Mode

```
(Windows I/O Subsystem Dispatcher code)
```

④

⑤

Your thread doesn't  
block here; it  
keeps running! ⑤

The CLR's Thread Pool

Threads can extract  
completed IRP's  
from here

NTFS Driver

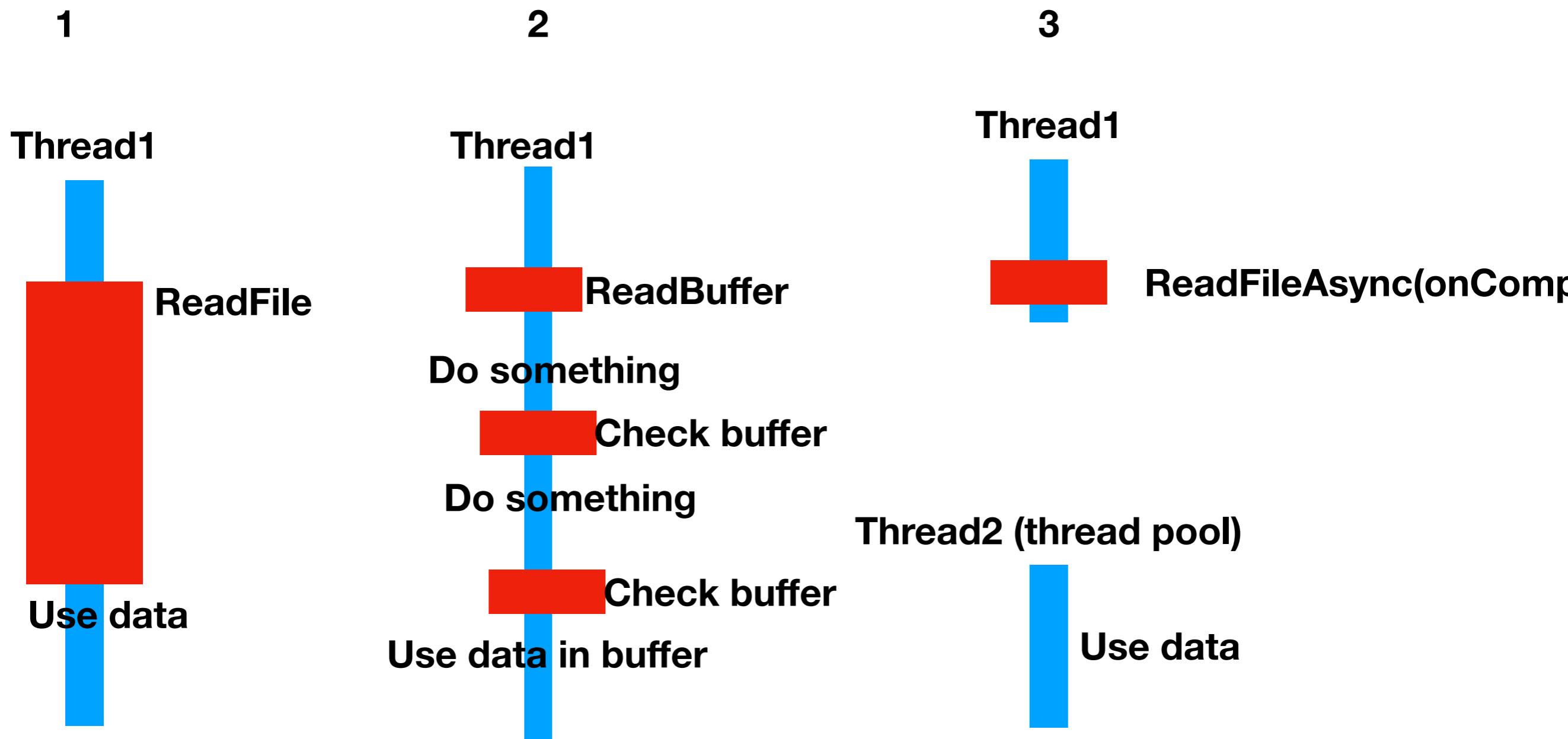
IRP  
Queue



⑧

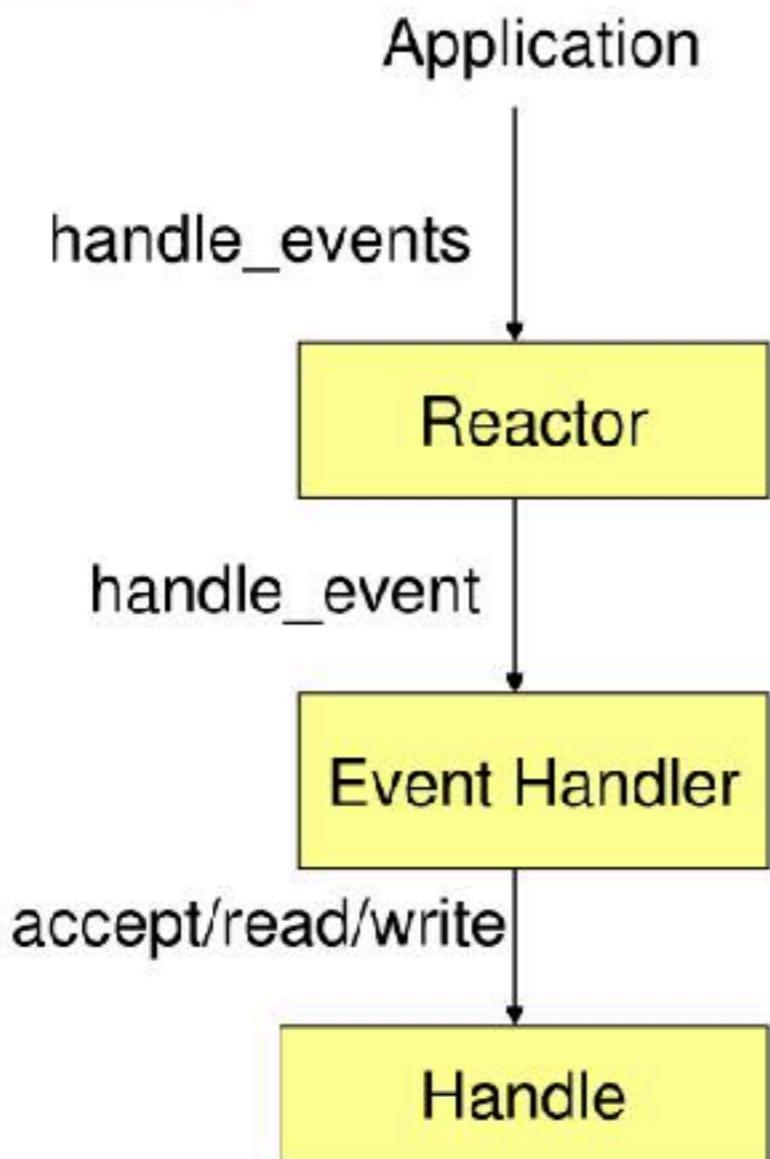
b

c

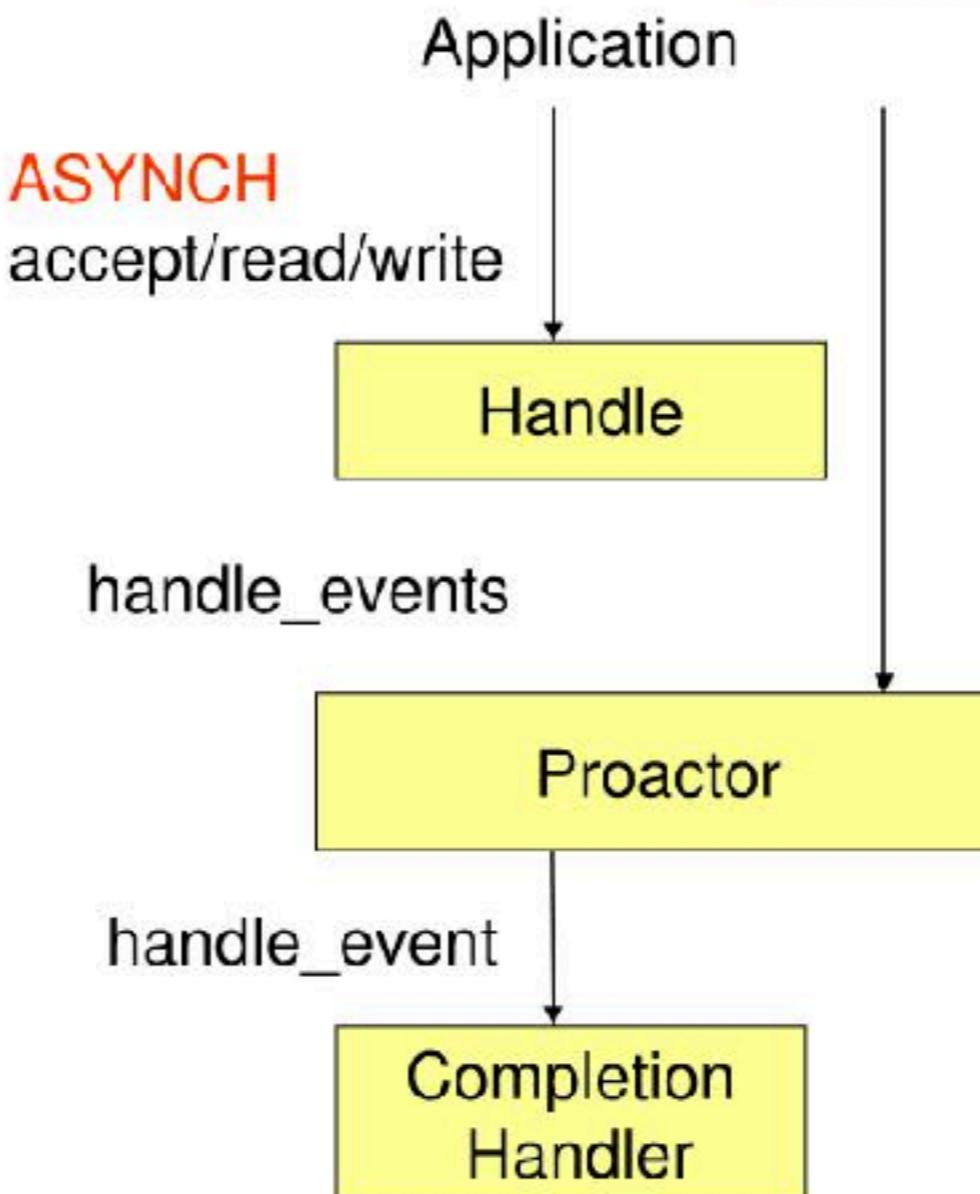


# Compare Reactor vs. Proactor Side by Side

Reactor



Proactor



Reactor and Proactor are two event handling patterns

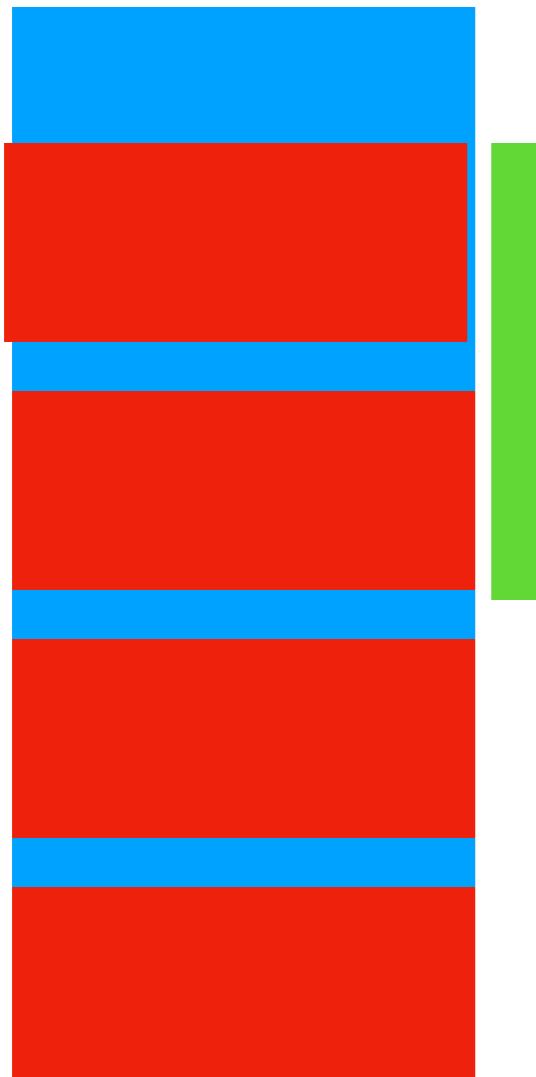
Proactor pattern is an asynchronous version of the Reactor pattern.

# *Lock-free updates*

`x = x * 10;`



# Atomic Operation



**data ++;**

1. Load from RAM into CPU register
2. Increment CPU register
3. Store from CPU register into RAM.



Reading and writing 64-bit fields is non atomic on 32-bit environments because it requires two separate instructions: one for each 32-bit memory location.

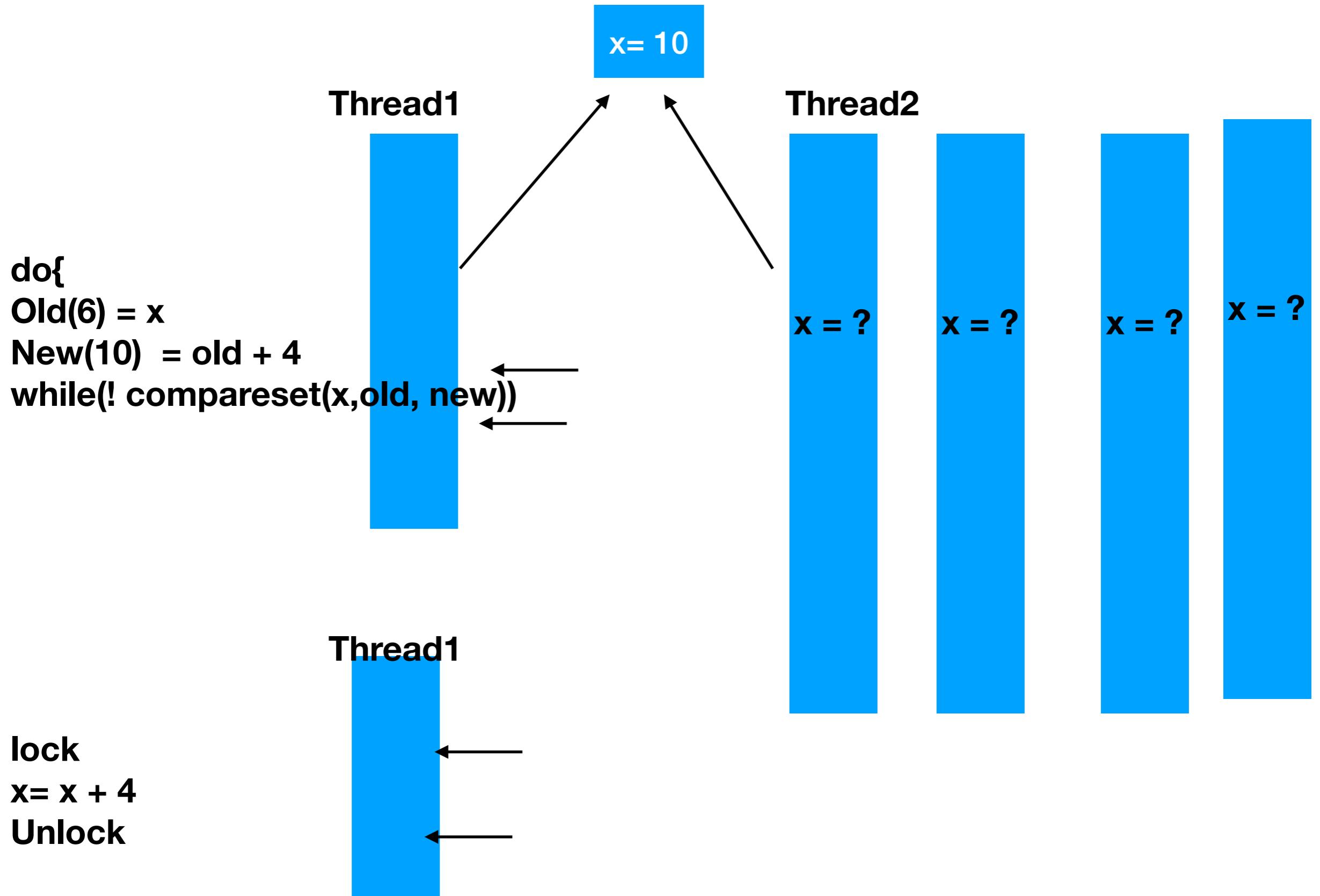
# Interlocked

- Increment
- Decrement
- Add
  - value can be a negative number allowing subtraction
- Read
  - performs an atomic Read on a 64-bit field:
- `old val = Exchange(ref _sum, new val)`
  - Write a 64-bit field while reading previous value
- `CompareExchange(ref _sum,new val, old val)`
  - Update a field only if it matches a certain value
  - enables lock-free read-modify-write operations

# *Lock-free updates*

1. Take a “snapshot” of  $x$  into a local variable
2. Calculate the new value
3. Write the calculated value back if the snapshot is still up-to-date using `Interlocked.CompareExchange`
4. If the snapshot was stale, *spin* and return to step 1

## Optimistic vs pessimistic



# Thread-safe collections

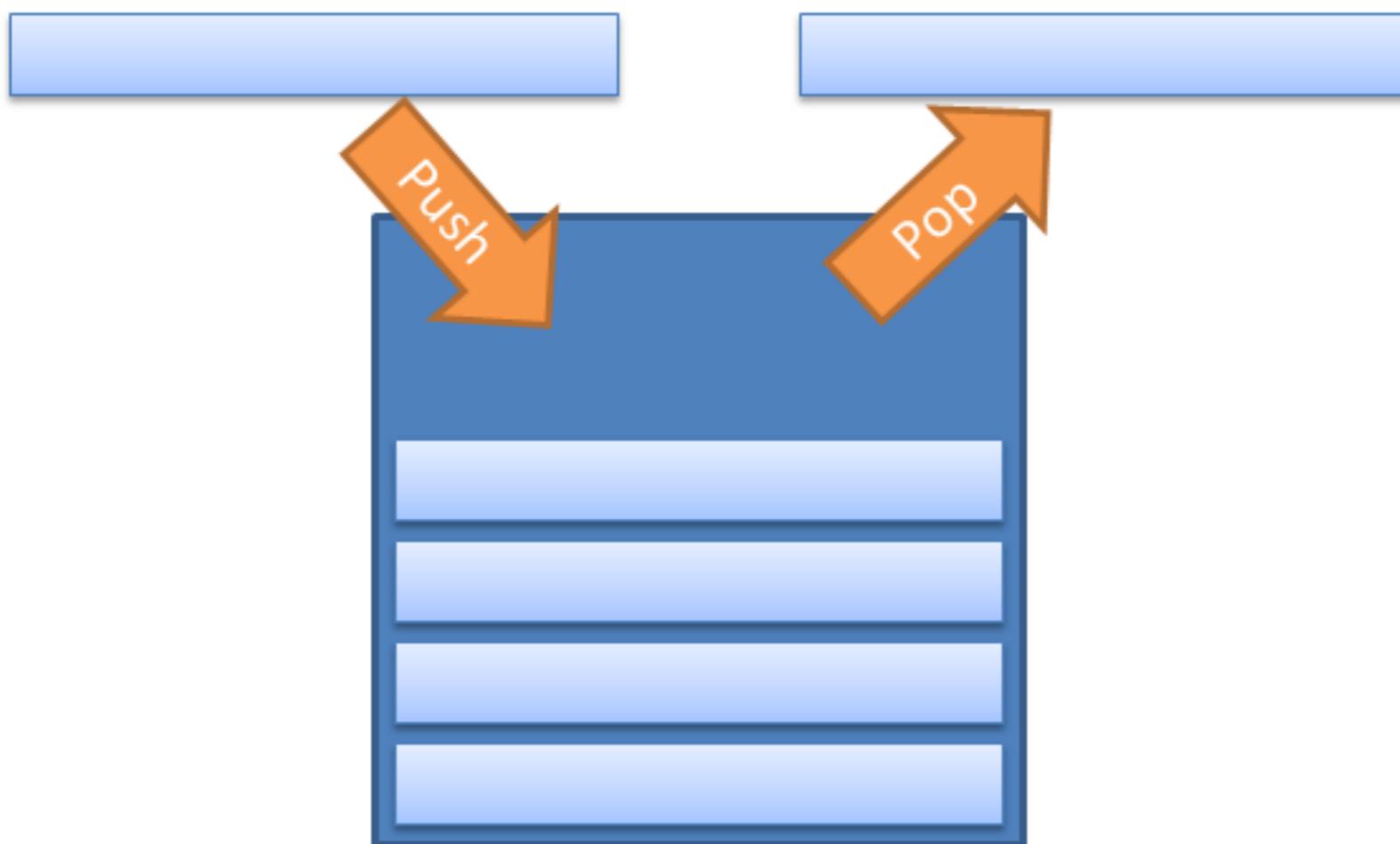
`ConcurrentBag<T>`

`ConcurrentStack<T>`

`ConcurrentQueue<T>`

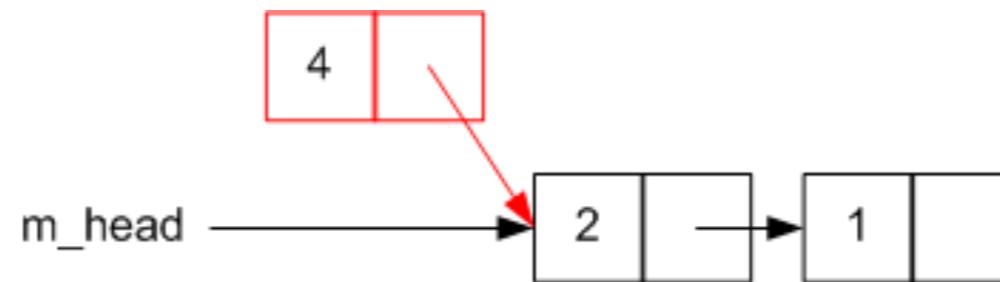
`ConcurrentDictionary<TKey, TValue>`

# Concurrent Stack

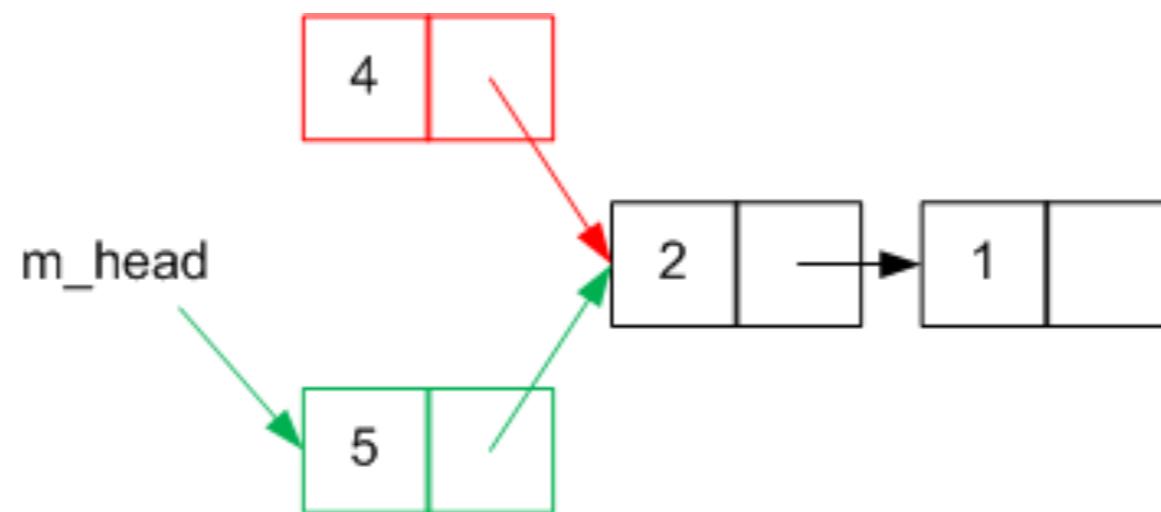


# Adding a item to stack

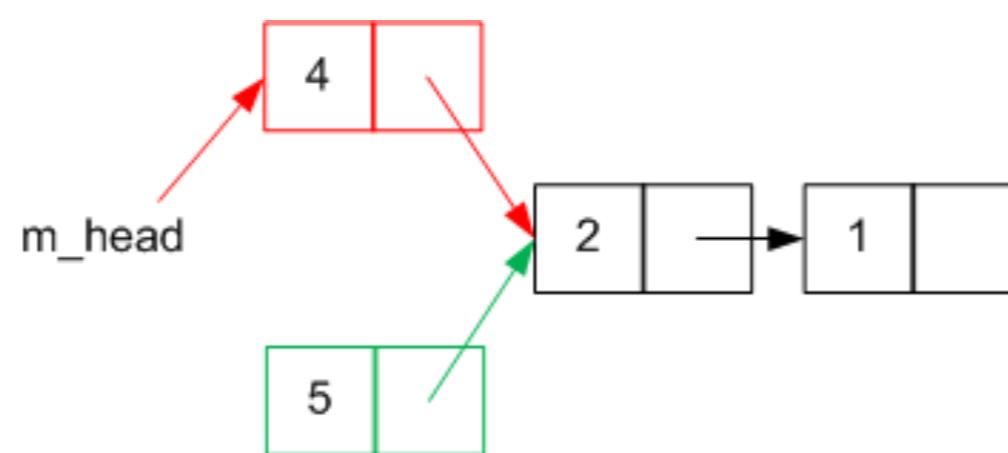
Thread 1, cycle 1 ,  
executes 1,2



Thread 2, cycle 2 ,  
executes 1,2,3

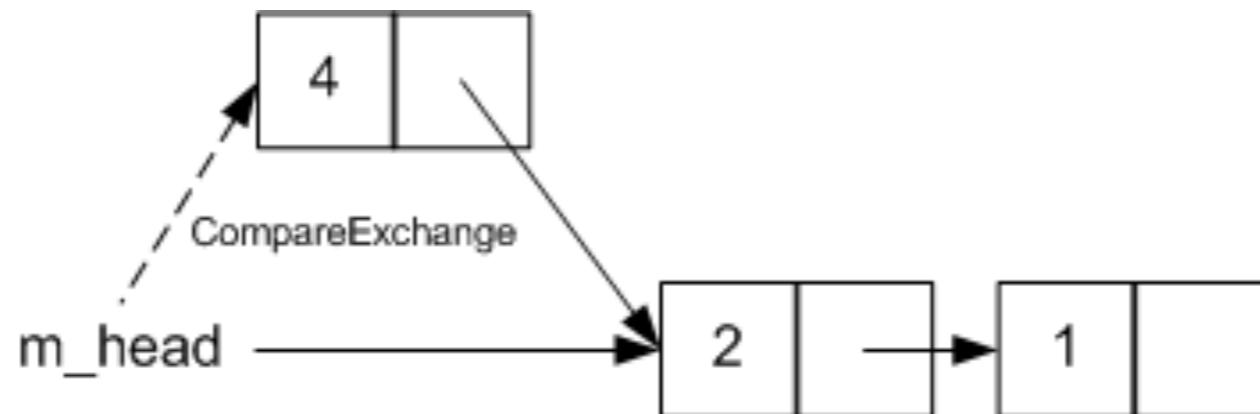


Thread 1, cycle 3 ,  
executes 3



# Adding a item to Concurrent Stack

```
public void Push(T item)
{
    Node node = new Node(item);      //step 1
    do {
        node.m_next = m_head;      //step 2
    } while (
        Interlocked.CompareExchange(ref m_head, node, node.m_next) !=
        node.m_next);               //step 3
}
```



Perform step 3 *only if the head hasn't changed in the meantime*. If it has been changed, then go back to step 2 to use the updated head node as the value of m\_next.

# ConcurrentQueue

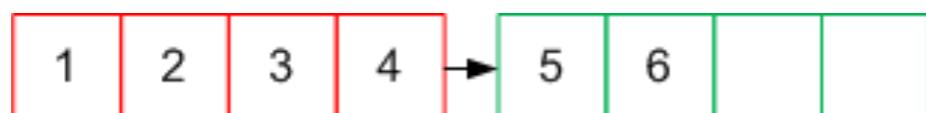


# ConcurrentQueue



*m\_head*  
*m\_tail*

three items are enqueued



*m\_head*

*m\_tail*

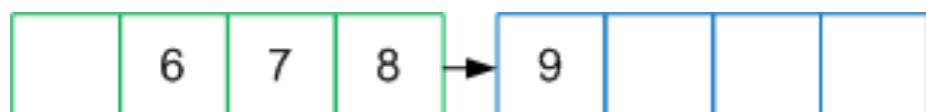
Three more items are then enqueue



*m\_head*

*m\_tail*

And another three



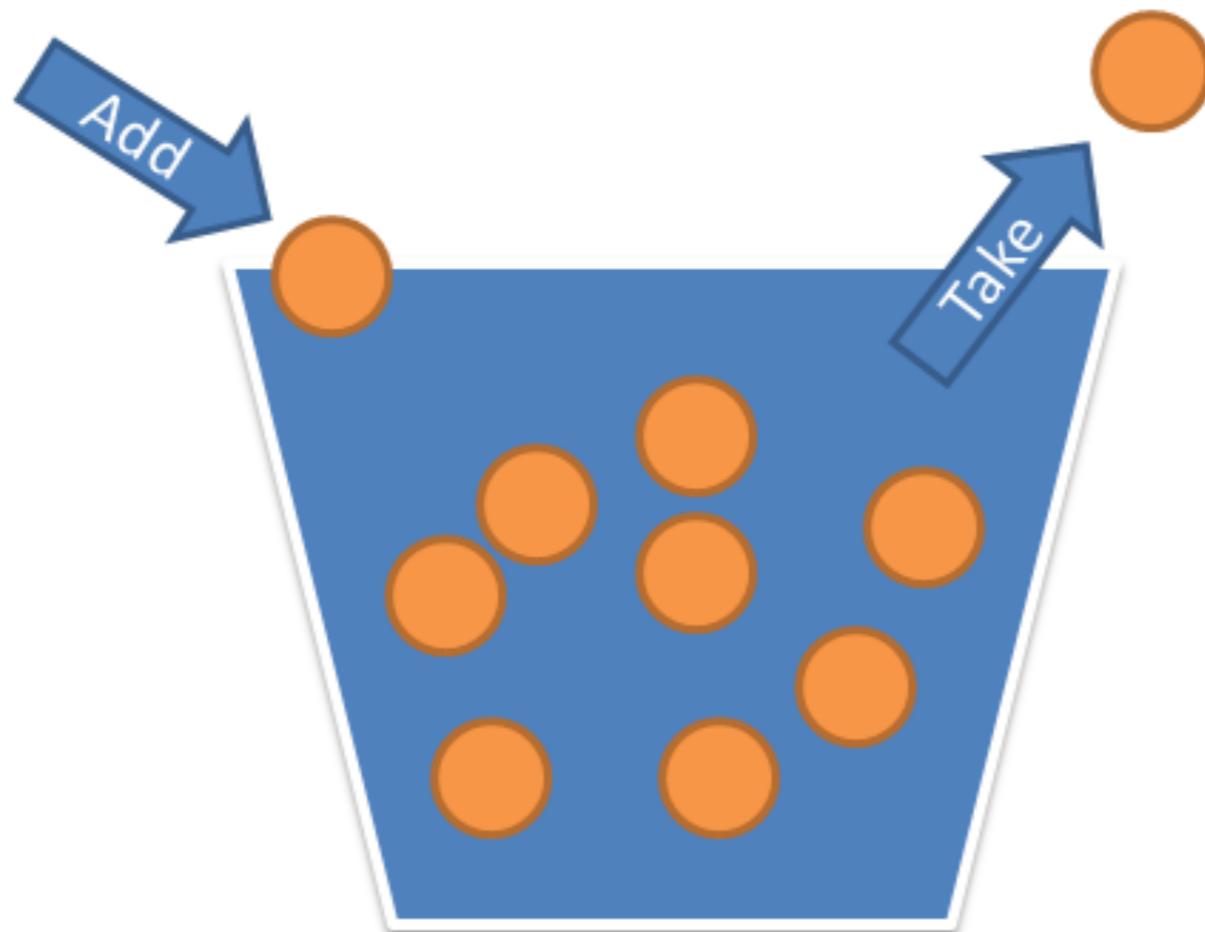
*m\_head*

*m\_tail*

Five items are then dequeued from the head.

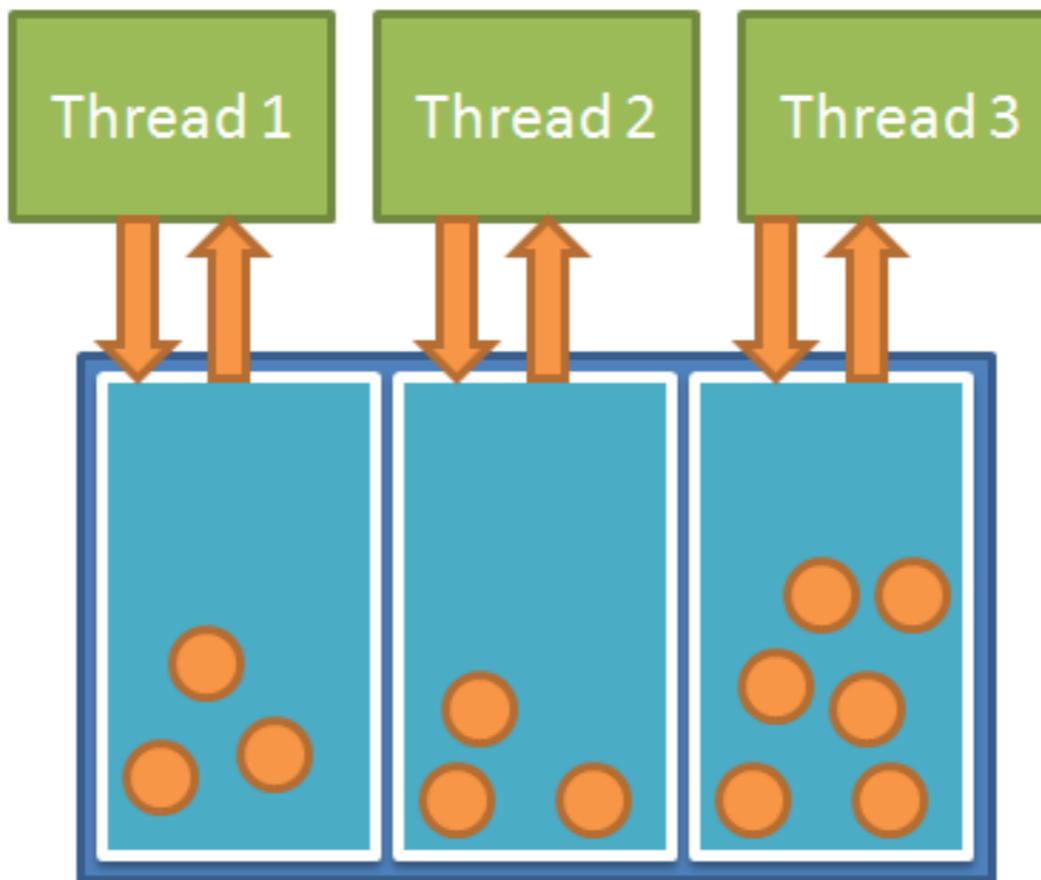
# ConcurrentBag

**ConcurrentHashMultiset**



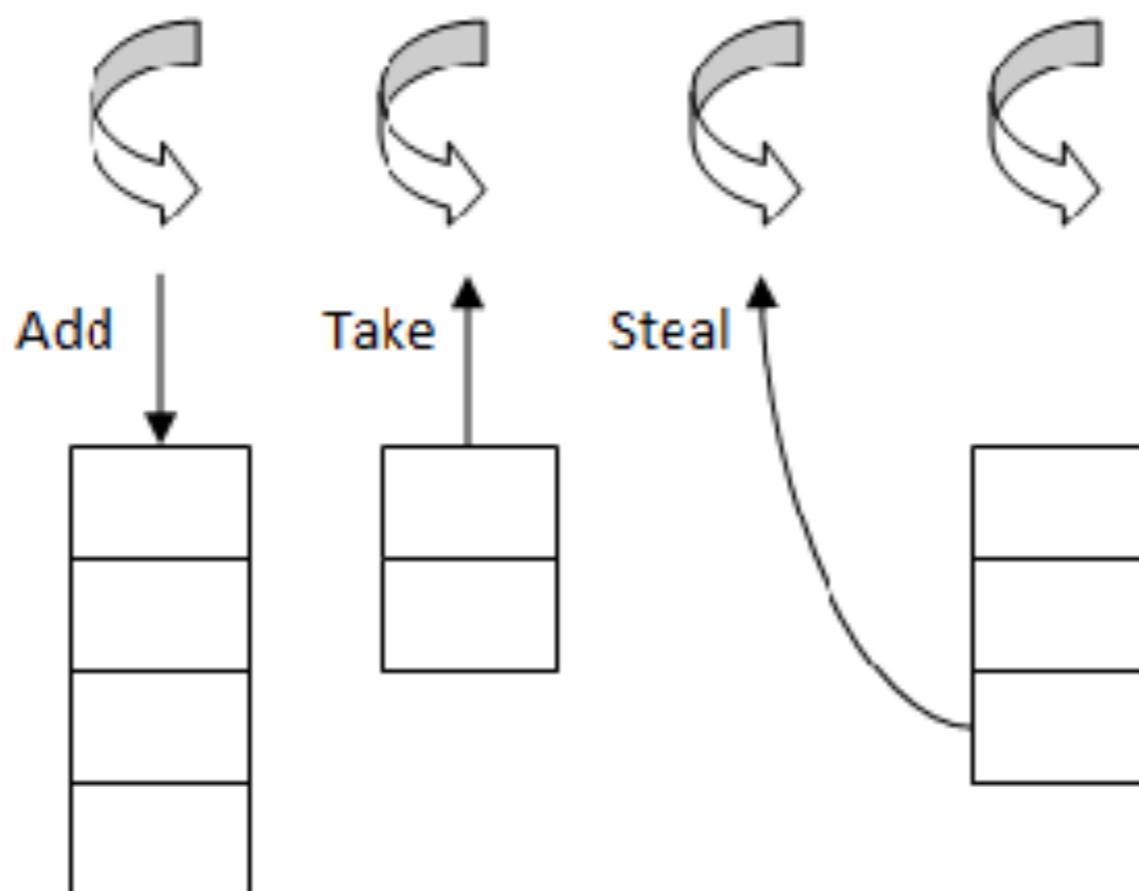
a thread-safe, unordered collection of objects

# concurrent bag



if local queue is empty, it will then perform some locking and look at one of the queues from another thread.

# concurrent bag



```
static void Main()
{
    bool complete = false;
    var t = new Thread(() =>
    {
        int i=0;
        while (!complete)
            i++;
        Console.WriteLine(i);
    });
    t.Start();
    Thread.Sleep(1000);
    complete = true;
    t.Join();
}
```

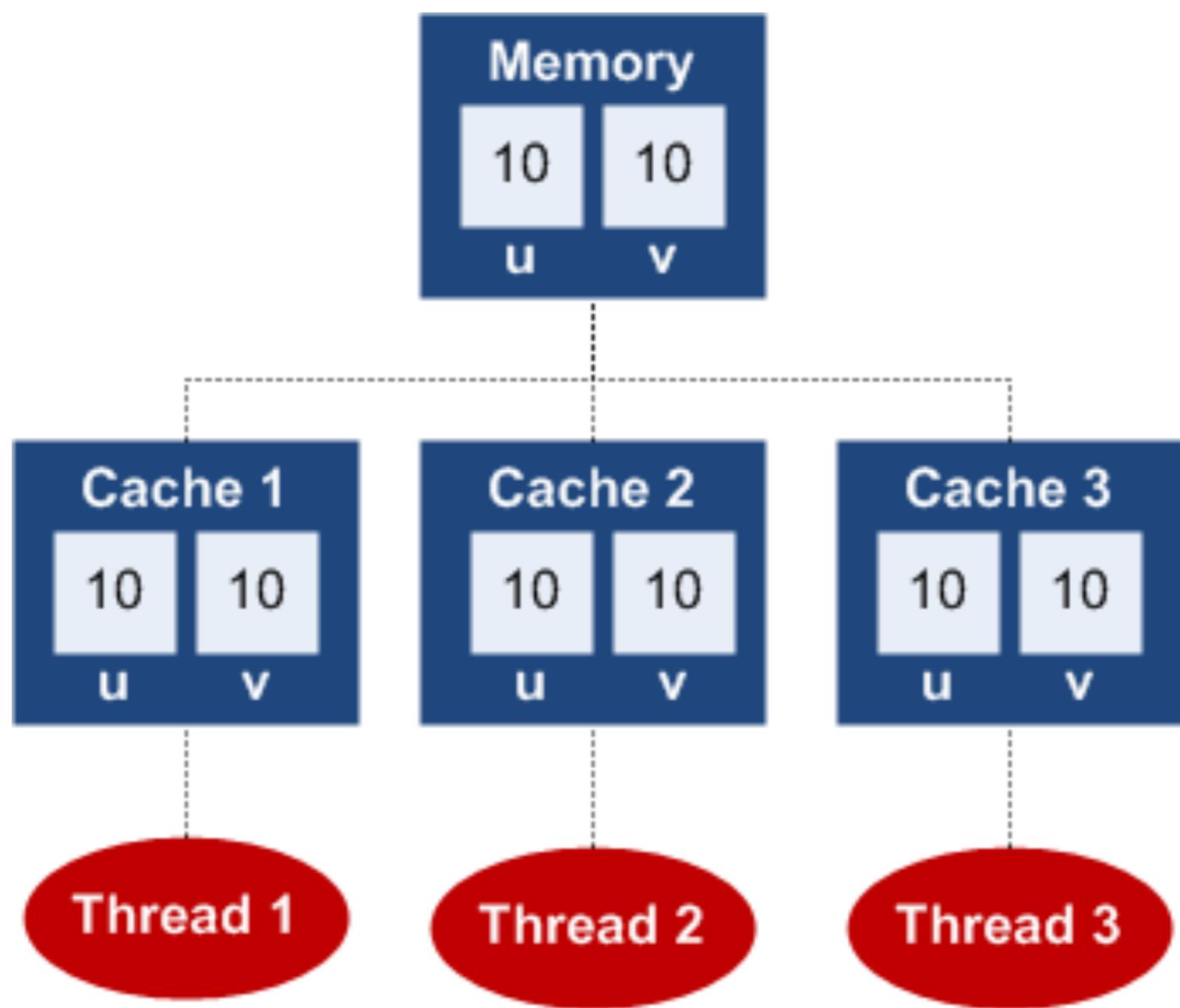
```
class Foo
{
    int _answer;
    bool _complete;
    void A()
    {
        _answer = 123;
        _complete = true;
    }
    void B()
    {
        if(_complete)
        {
            Console.WriteLine (_ answer);
        }
    }
}
```

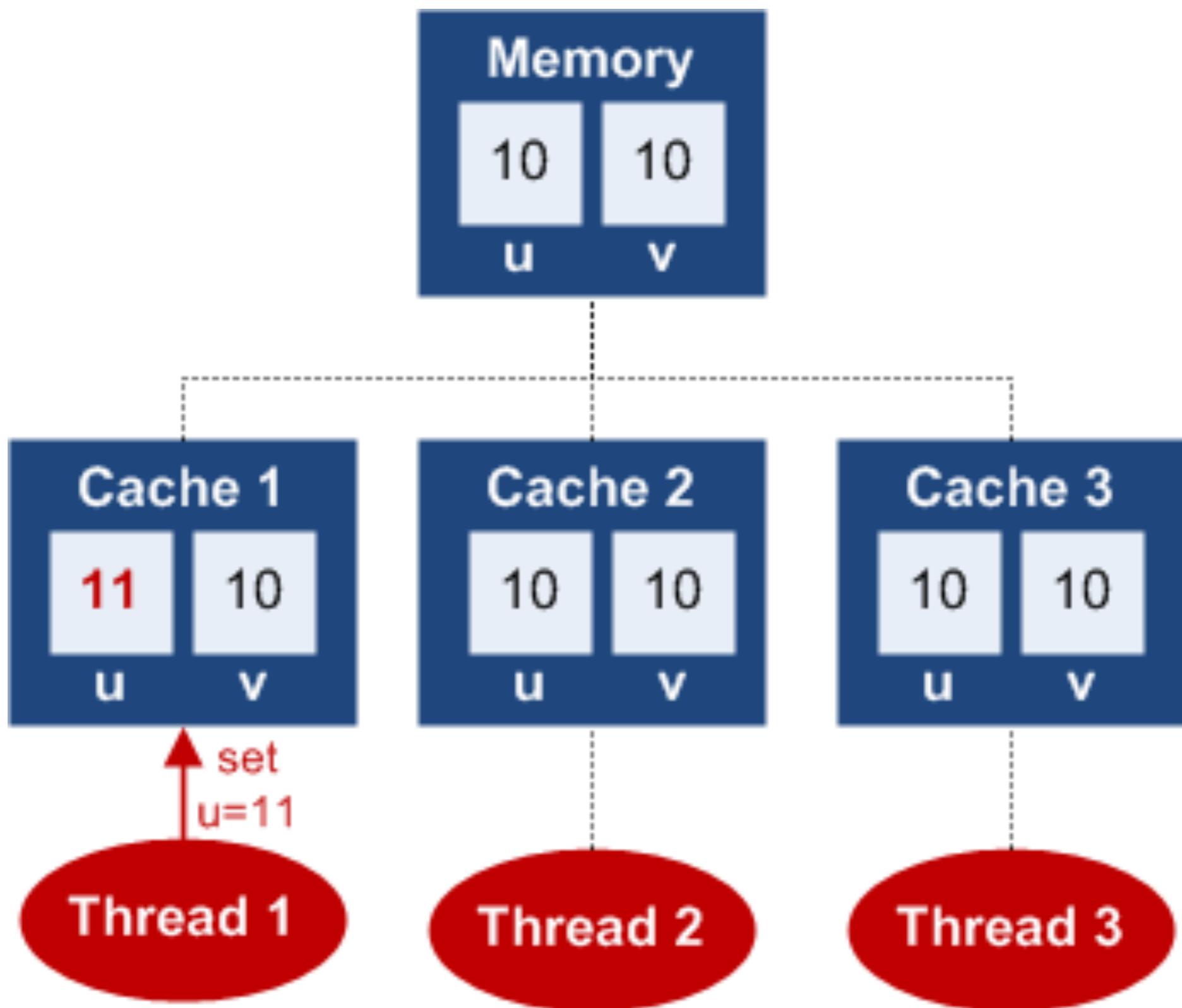
```
class Foo
{
    int _answer;
    bool _complete;
    void A()
    {
        _answer = 123;
        _complete = true; //← instructions could get reordered causing B to write
        "0"
    }
    void B()
    {
        if (_complete)
            Console.WriteLine (_answer);
    }
}
```

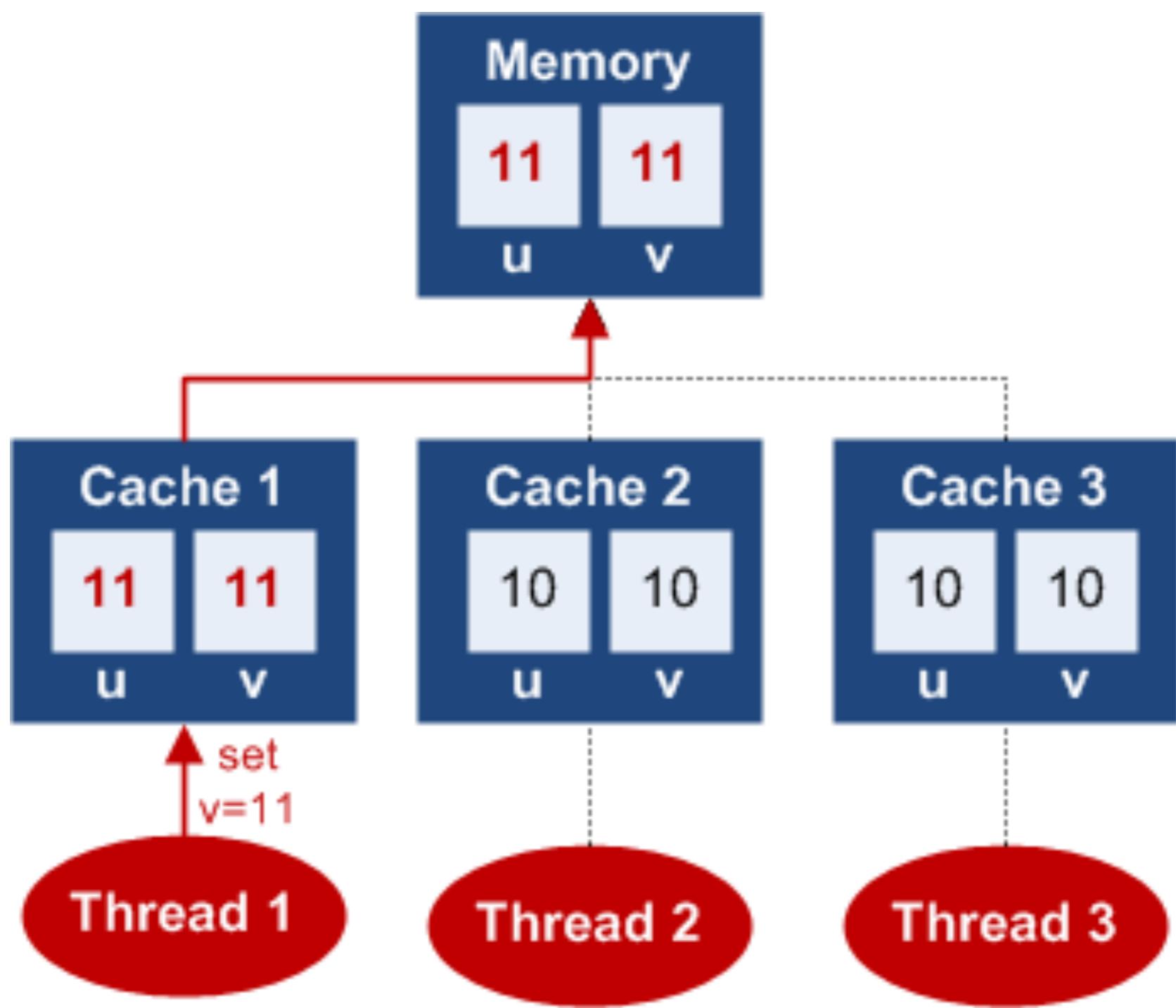
```
static void Main()
{
    bool complete = false;
    var t = new Thread(() =>
    {
        int i=0;
        while (!complete)
        {
            Thread.MemoryBarrier();
            i++;
        }

        Console.WriteLine(i);
    });
    t.Start();
    Thread.Sleep(1000);
    complete = true;
    t.Join();
}
```

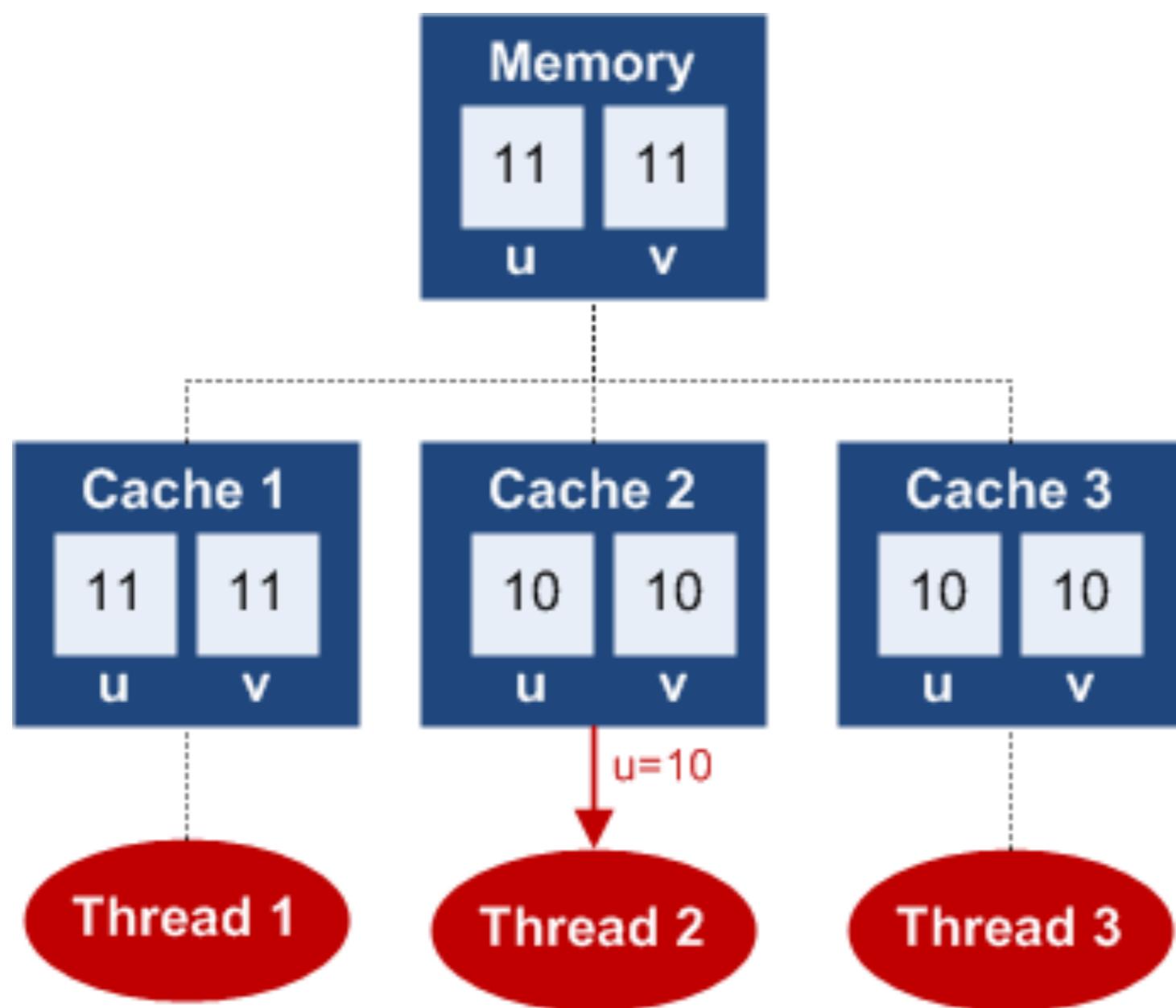
run it with optimizations enabled and Release Mode

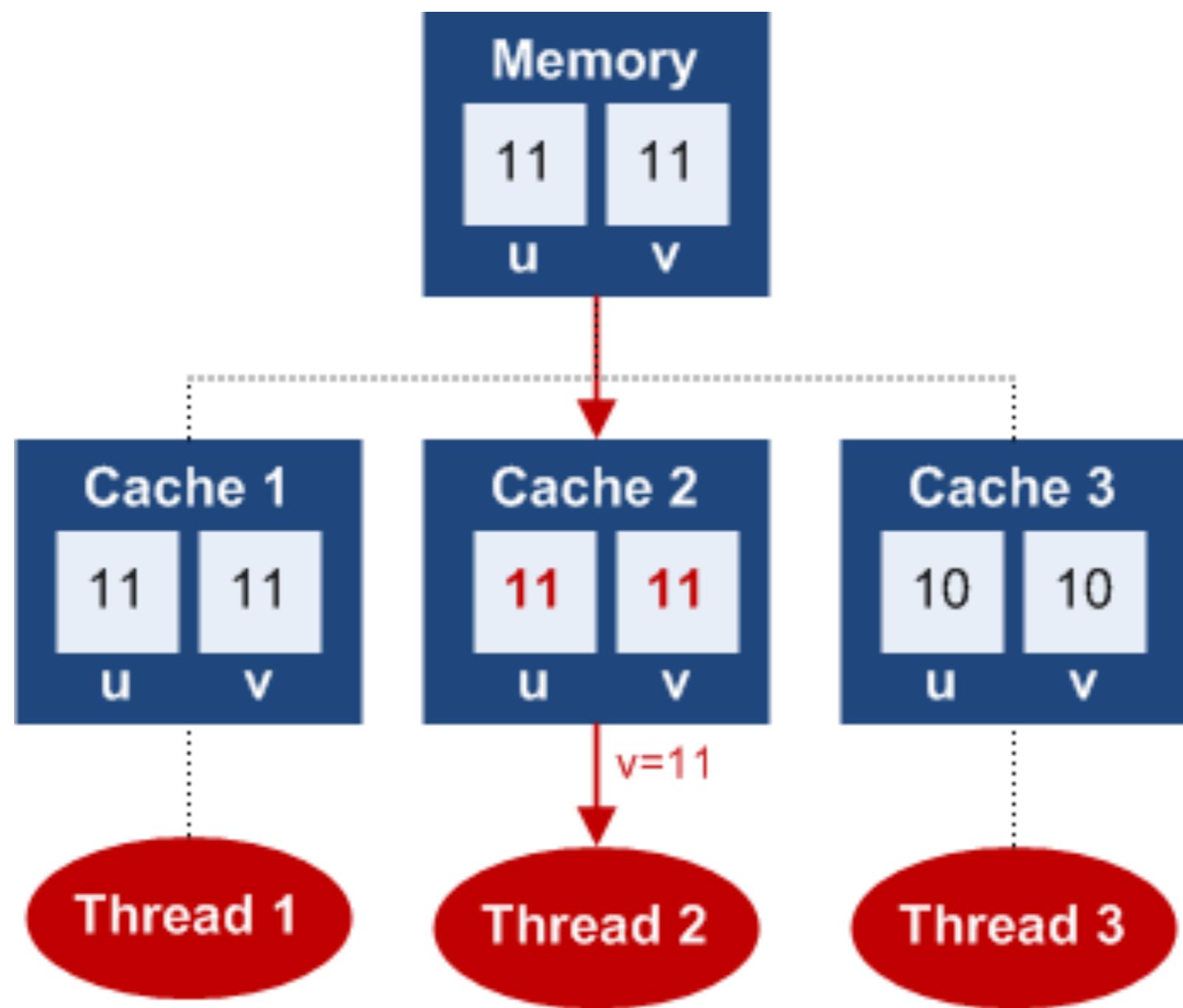






all C# writes are volatile



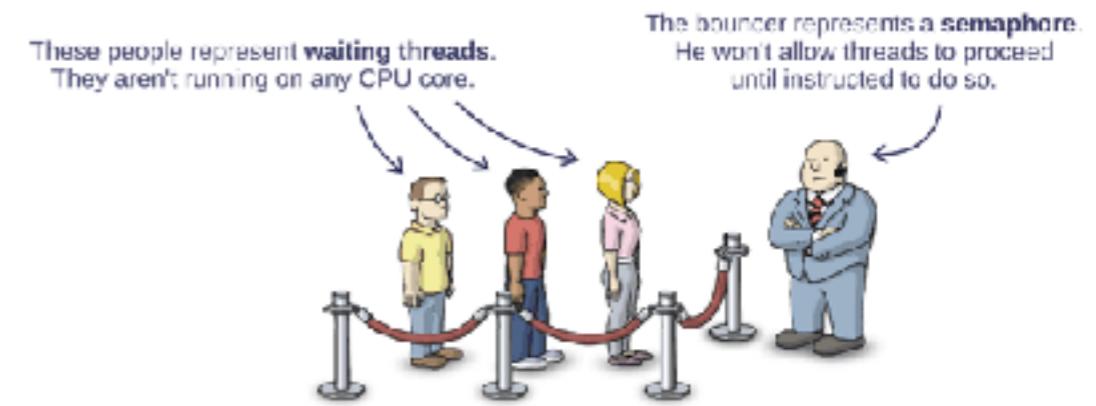


# Synchronization constructs

- **Simple blocking methods**
  - These wait for another thread to finish or for a period of time to elapse.
- **Locking constructs**
  - These limit the number of threads that can perform some activity or execute a section of code at a time.
- **Signaling constructs**
  - These allow a thread to pause until receiving a notification from another
- **Nonblocking synchronization constructs**
  - These protect access to a common field by calling upon processor primitives.

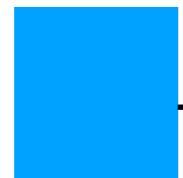


- **Nonblocking synchronization constructs**
  - volatile
  - Atomic.
- **Simple blocking methods (For a Period)**
  - Sleep (bad)
  - Join
  - Spin
- **Locking constructs**
  - *Exclusive* locking
  - nonexclusive locking
    - Semaphore
    - reader/writer locks.
- **Signaling constructs**
  - event wait
  - Latch
  - Barrier



**Subscribe to  
Keyboard events**

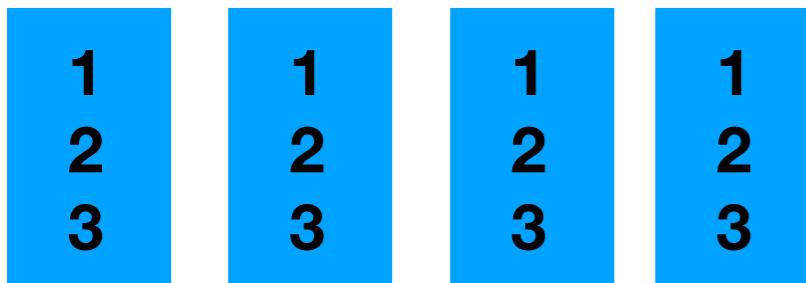
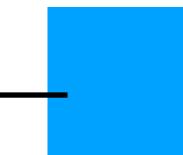
**Thread1**



**Buffer**

**Subscribe to  
Window events**

**Thread2**



**When user press Enter key,**

- 1# read all window events from buffer**
- 2# take a screen shot(image) of the desktop**
- 3# collect os info of active window**

**@ PID**

**@ title**

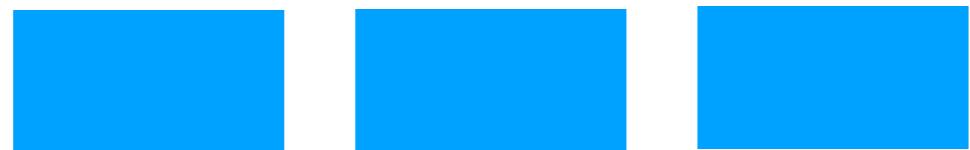
**@ type of app**

- 4# extract text from the image (OCR)**

- 5# mask the image**

- 6# store the image and data on the disk/nw**

**6 Async IO**



\* E

\*

\*

\*

\*

\*

\* C

\*

\*

\*

\*

\*

\* BP

\* ?

\*

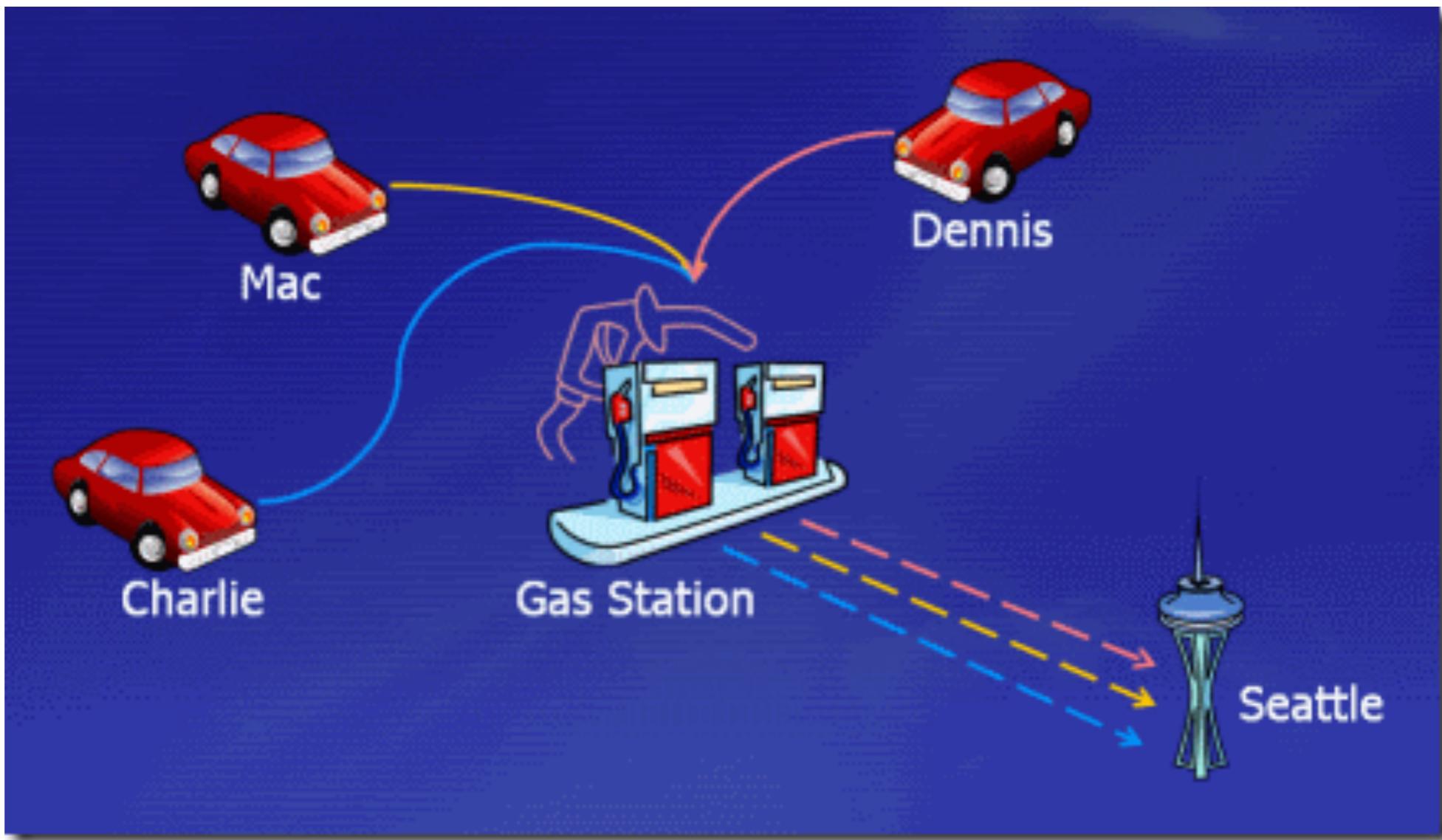
\*

\*

\*

\* BP

# Latch vs Barrier



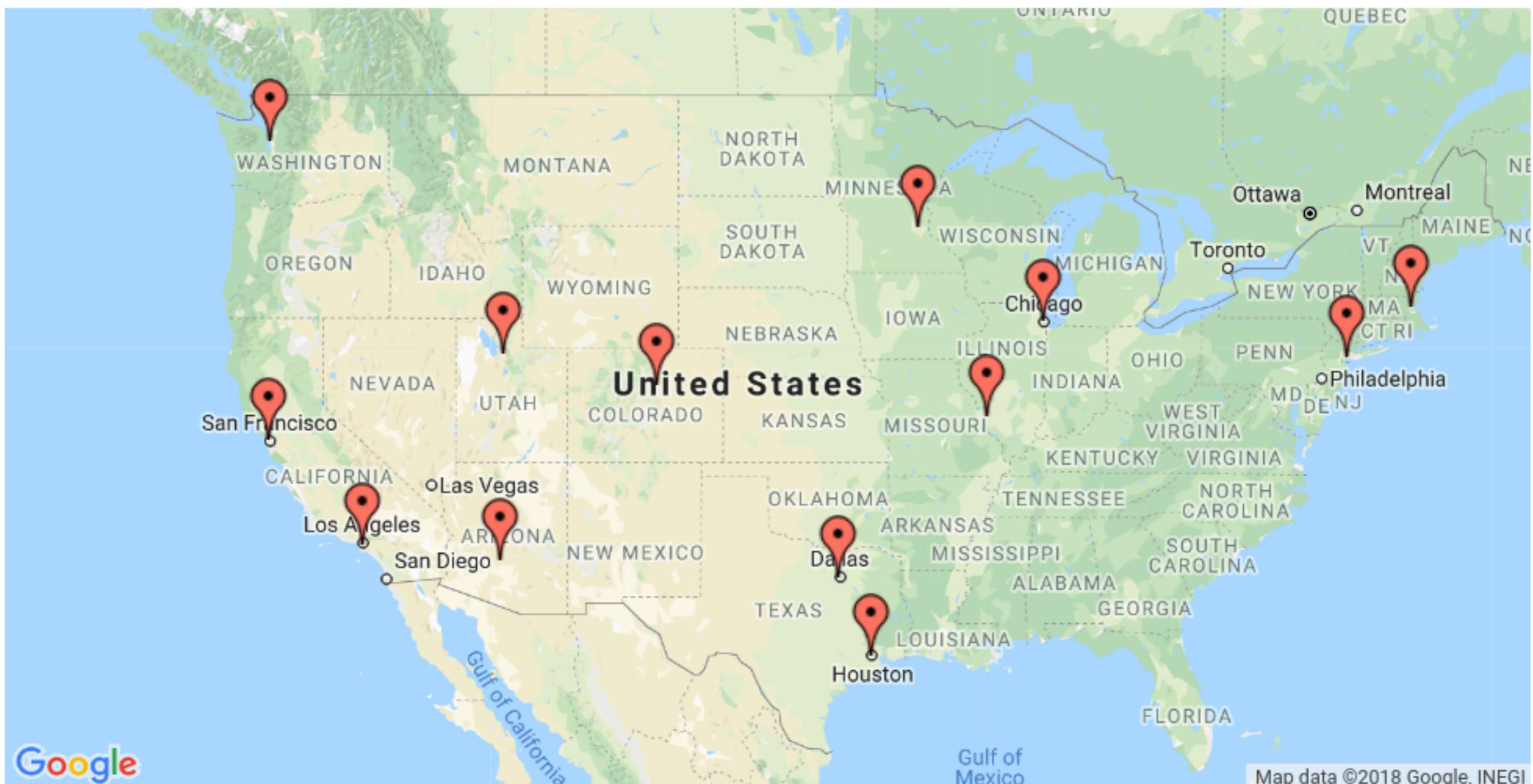
CountDownLatch, no. of calls(tasks) to the countdown() method

CyclicBarrier, no. of threads that should call await()



The deprecated Suspend and Resume  
methods have two modes: dangerous and  
useless!





0. New York - 1. Los Angeles - 2. Chicago - 3. Minneapolis - 4. Denver - 5. Dallas - 6. Seattle - 7. Boston - 8. San Francisco - 9. St. Louis - 10. Houston - 11. Phoenix - 12. Salt Lake City

```

static class DataModel {
    public final long[][][] distanceMatrix = {
        {0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972},
        {2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579},
        {713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260},
        {1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280, 987},
        {1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371},
        {1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999},
        {2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701},
        {213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099},
        {2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600},
        {875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162},
        {1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200},
        {2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504},
        {1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0},
    };
    public final int vehicleNumber = 1;
    public final int depot = 0;
}

```

- The number of vehicles in the problem, which is 1 because this is a TSP. (For a vehicle routing problem (VRP), the number of vehicles can be greater than 1.)
- The *depot*: the start and end location for the route. In this case, the depot is 0, which corresponds to New York.

**TSP with 10 cities can be solved by a DP method in almost 0.2 seconds using intel core i7. This number increases to almost 13 seconds (~60 times greater) with 15 cities.**