Multi Threading

» Skan.ai - chief Architect

» Ai.robotics - chief Architect

» Genpact - solution Architect

» Welldoc - chief Architect

» Microsoft

» Mercedes

» Siemens

» Honeywell

Mubarak

Agenda

- Concurrency
- Locks
- Best Practices
- Anti Patterns
- Debugging

Parallelism

Task Parallelism

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

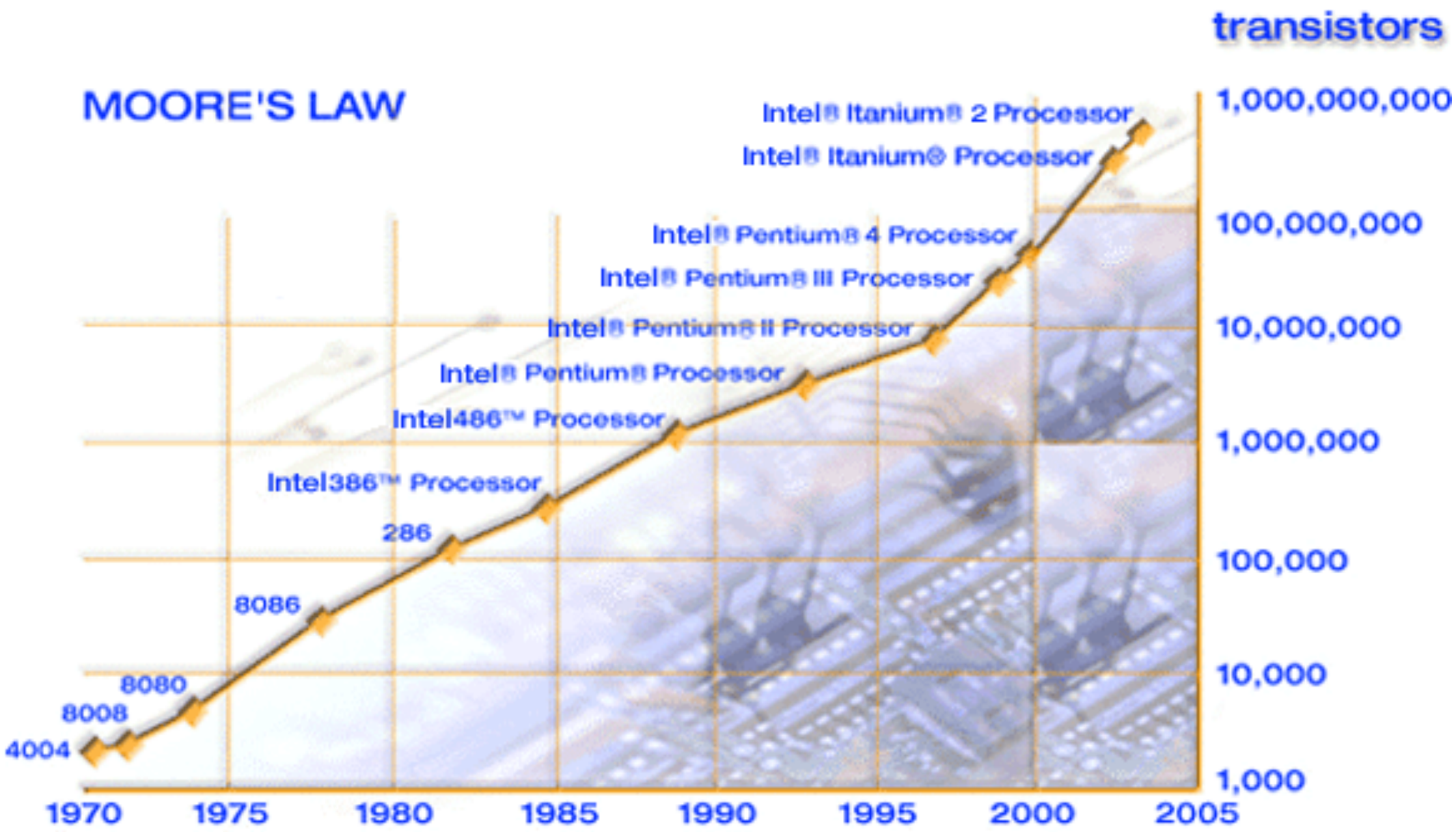
Data parallelism

- 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.

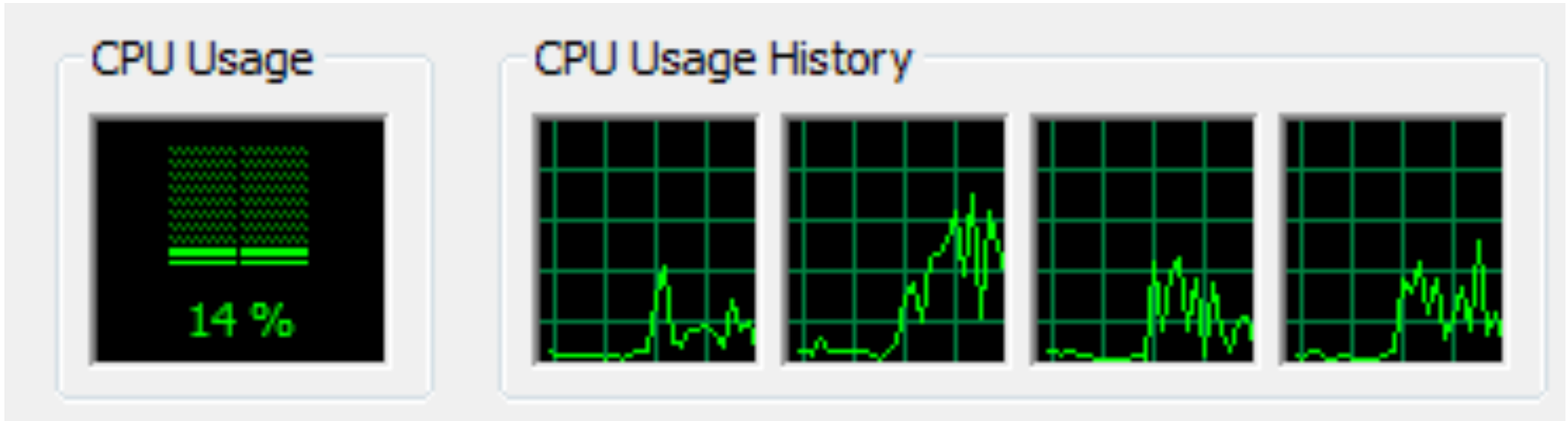
Example

```
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);
}
```

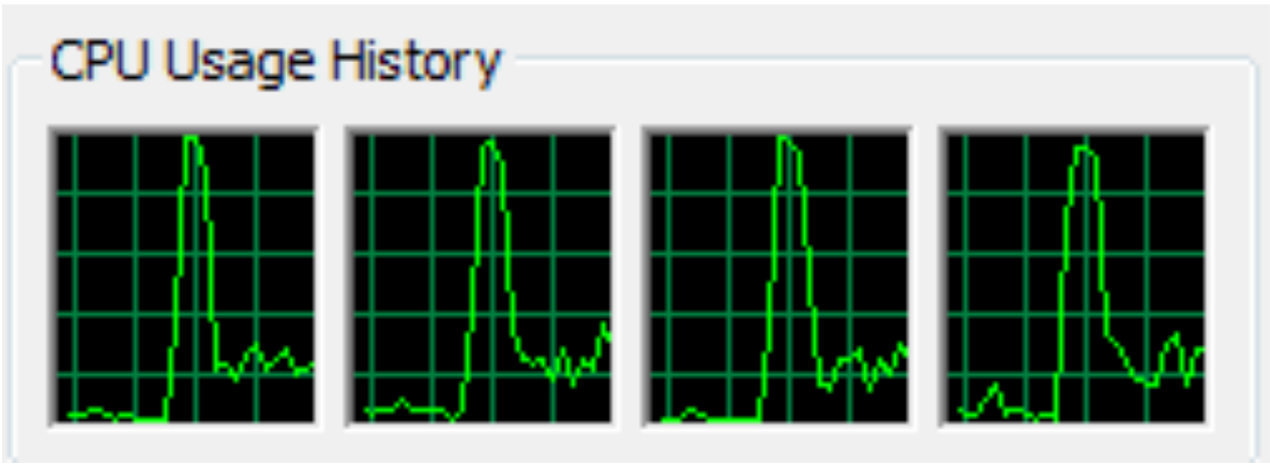


With four cores : over 12 seconds

Parallel Programming

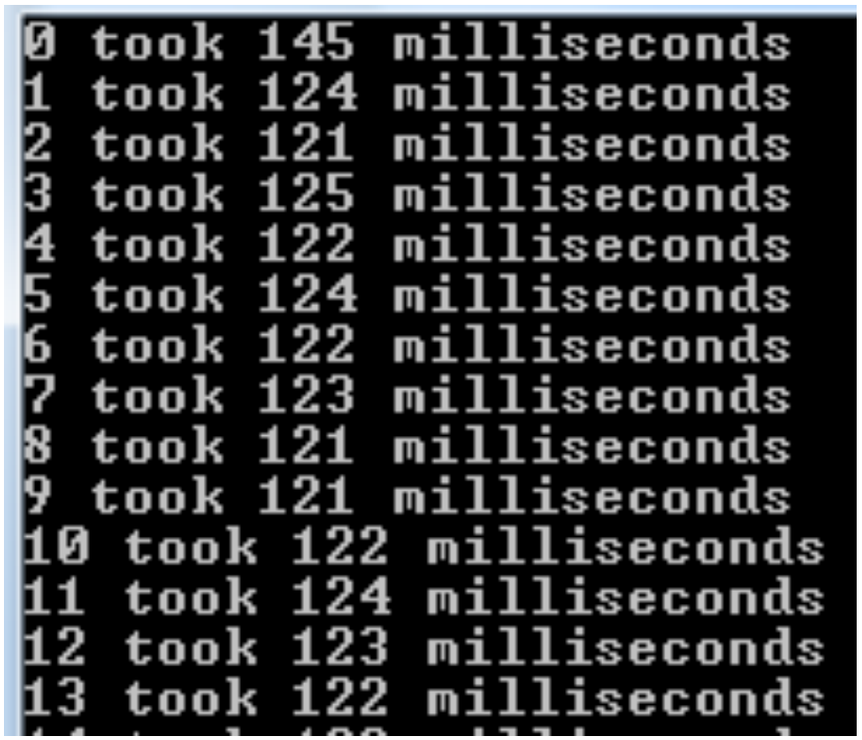
```
//for (int i = 0; i < 100; i++)

Parallel.For(0, 100,
 delegate(int i){    doWork(i);      });
```



The process finishes in 3.7 seconds instead of 12.

Parallel Programming



Single-threaded

Individual tasks may take a little longer

Parallel

some as high as 400-milliseconds

Foreground Threads
Vs
Background Threads

Try to avoid using foreground threads as much as possible

When you really want to complete, like flushing data from a memory buffer out to disk.

Cost of a Memory

- x86, 700 bytes.
- x64, 1,240 bytes.
- IA64 CPUs, 2,500 bytes

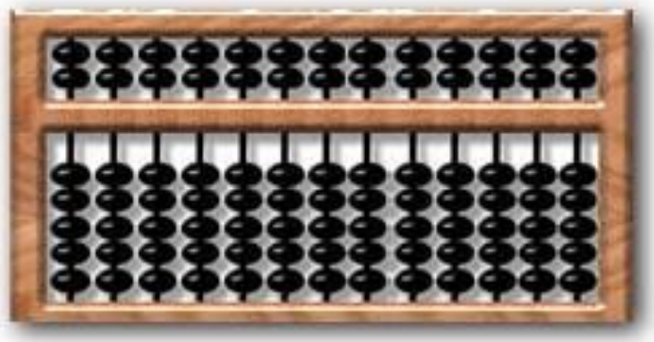
Cost of a Thread

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

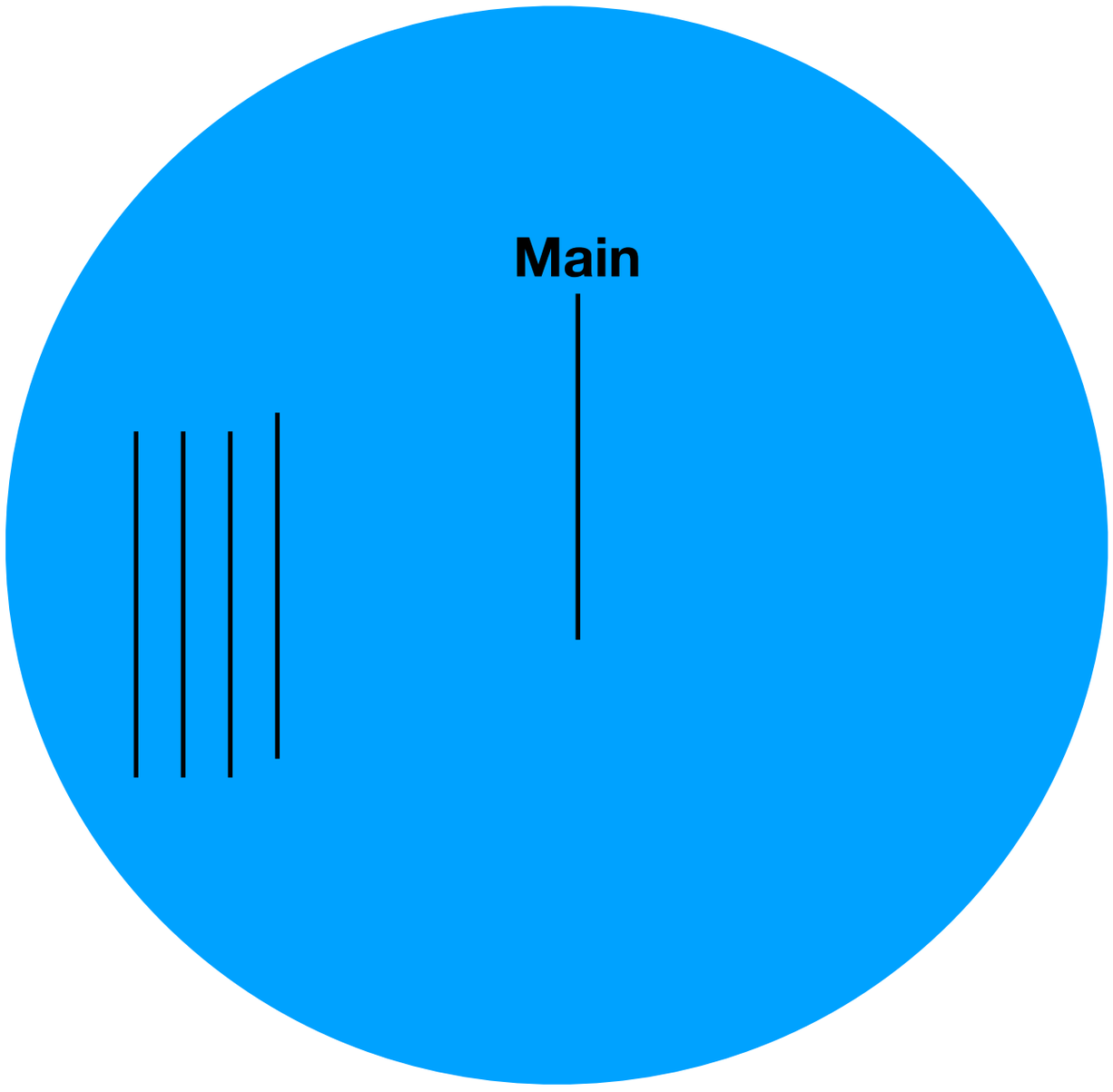
Don’t create Threads

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 ?


Main

CARPOOLING

Because Gas is Just Too Expensive

When To Create Thread

Do you require a foreground thread

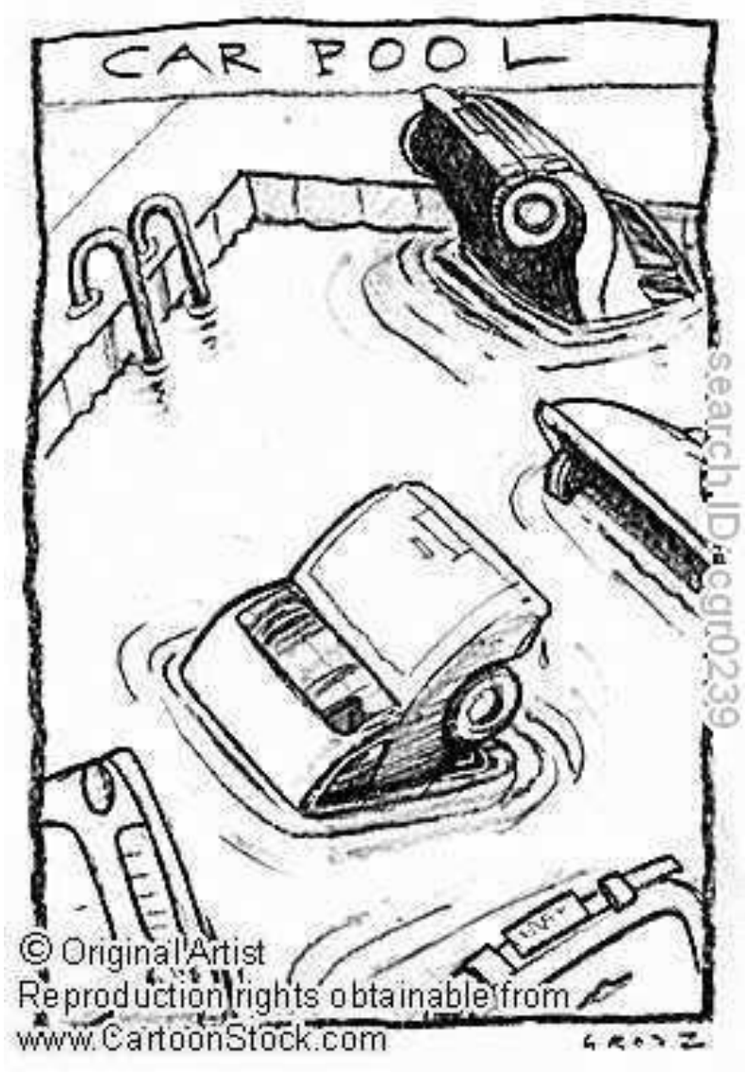
When To Create Thread

Change priority

When To Create Thread

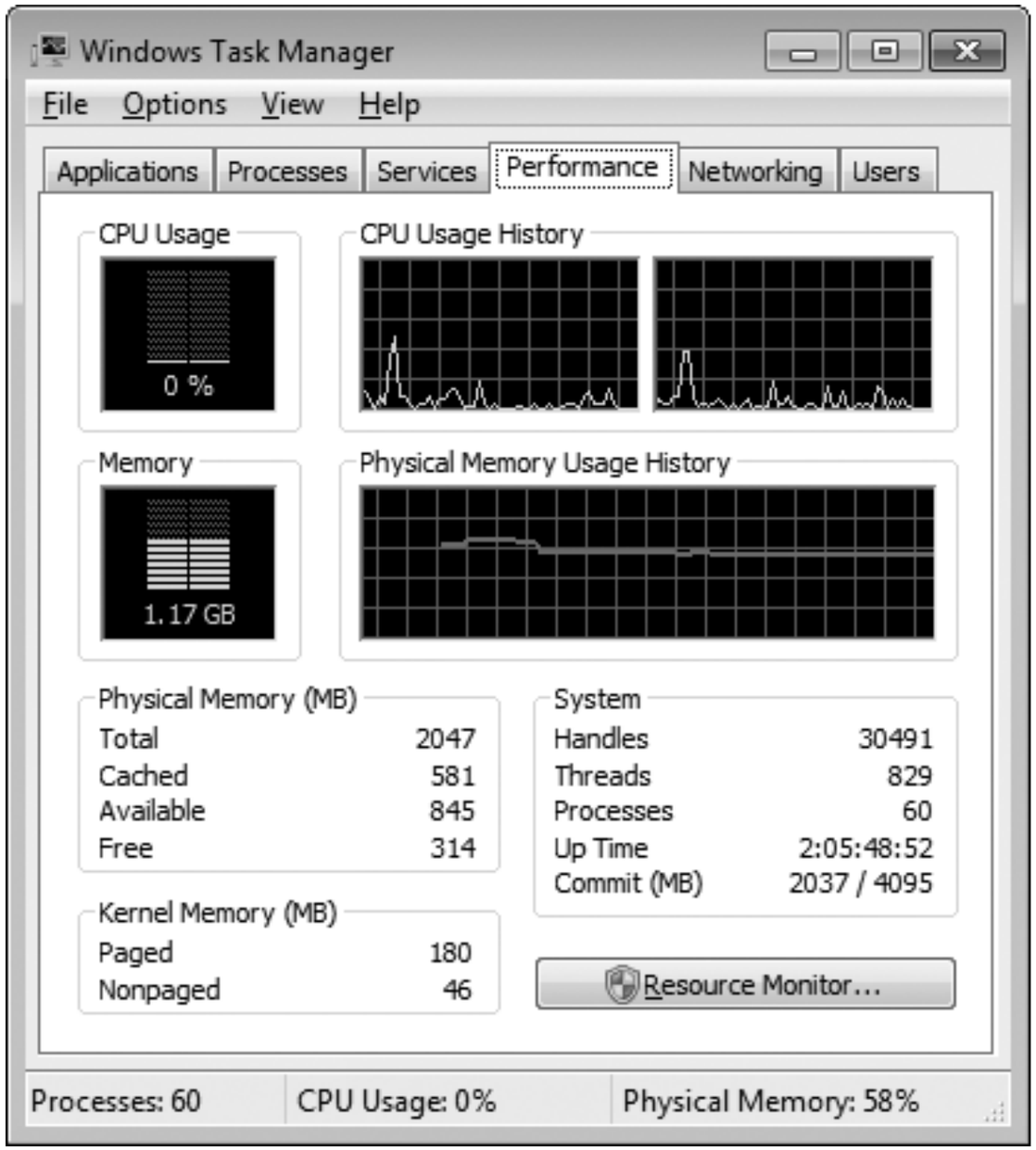
Long running

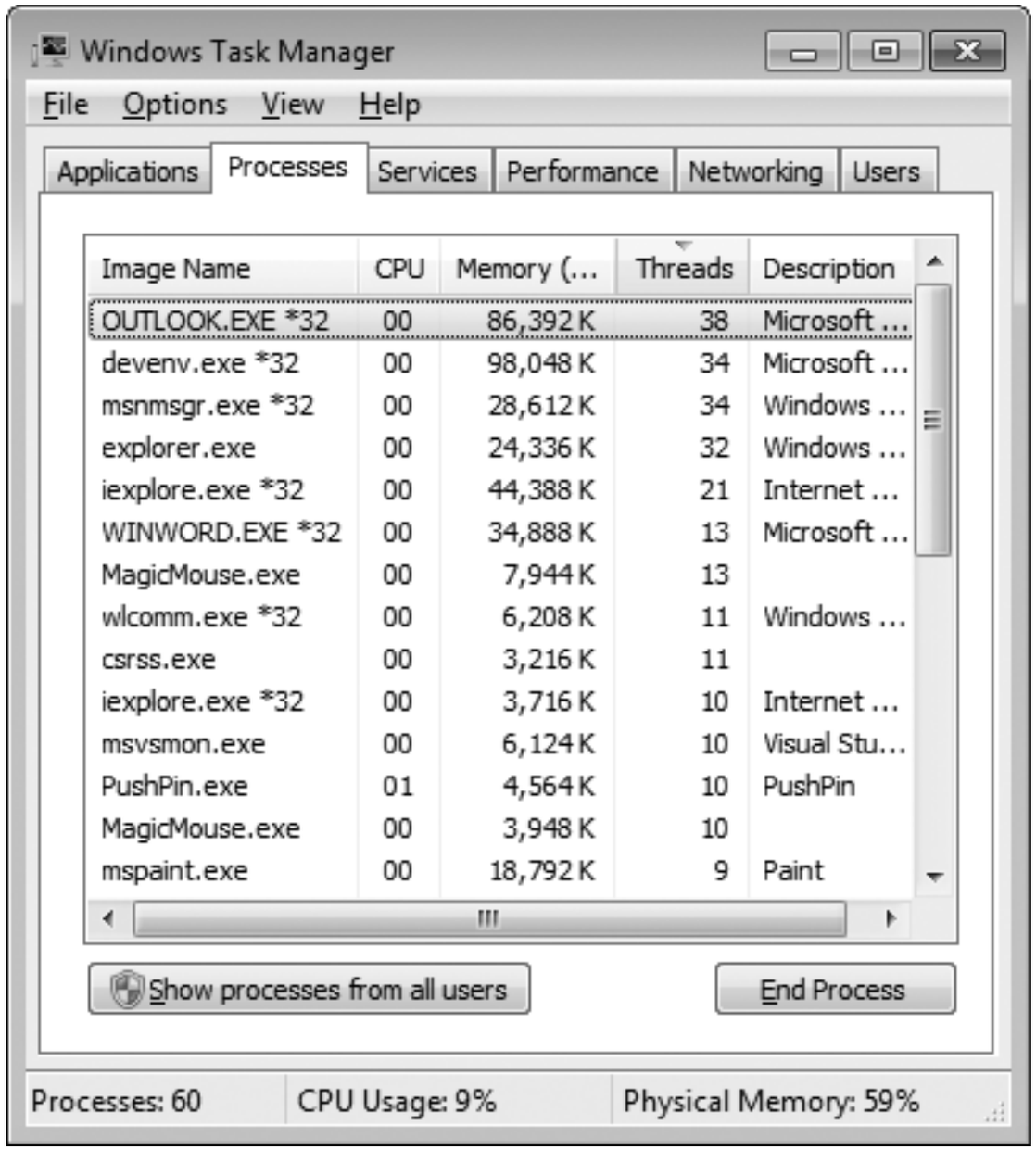
When To Create Thread



You need to have a stable identity associated with the thread

Do these applications need all these threads





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

Max : 5 threads

```
executor.execute(
executor.execute(
executor.execute(
executor.execute(
executor.execute(
```

Memory reader

Memory reader

Memory reader

Memory reader

Memory reader


```
executor.execute(
```

Memory writer

```
executor.execute(
executor.execute(
executor.execute(
executor.execute(
executor.execute(
executor.execute(
```

New work request

Worker Thread is Available

Execute Callback

Worker Thread is NOT Available

Running Threads >= Max Threads

Wait for running thread to complete

Running Threads < Max Threads

Default = CPU's Count * 250 and Climbing

Running Threads < Min Threads

Default = CPU's Count

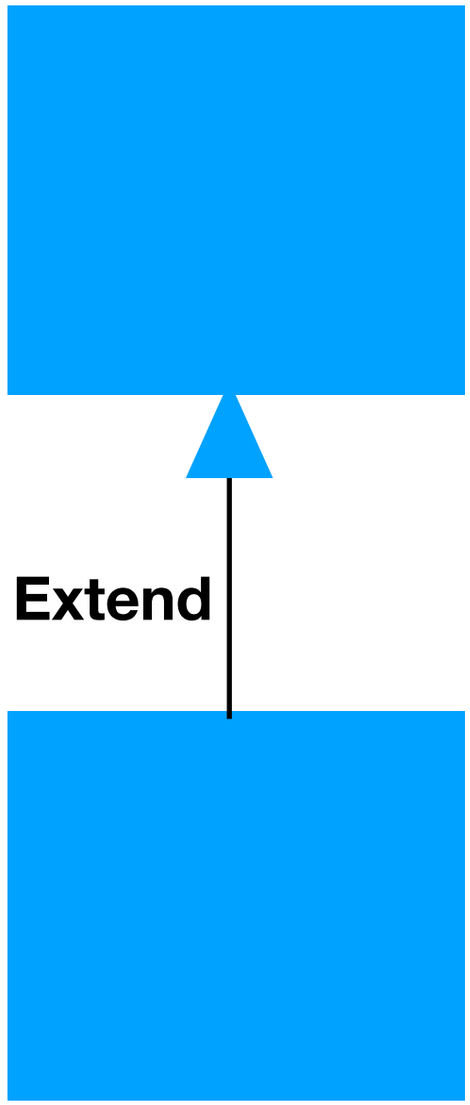
Running Threads >= Min Threads

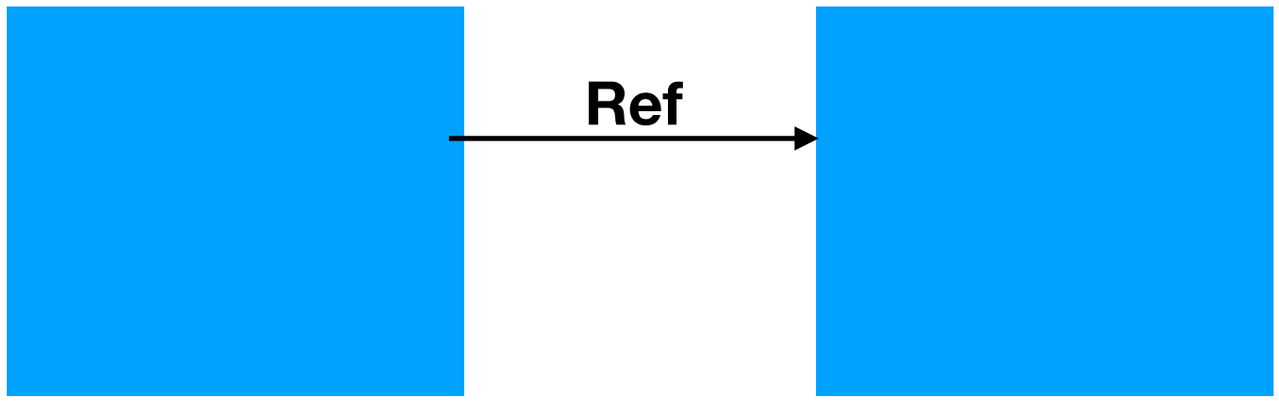
Suspend Thread Creation

«invariant»
{Creation Rate = 2 Threads Per Sec}

Create New Thread

Execute Callback


Extend


Ref

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  }
```

System.Threading

Parallel.For

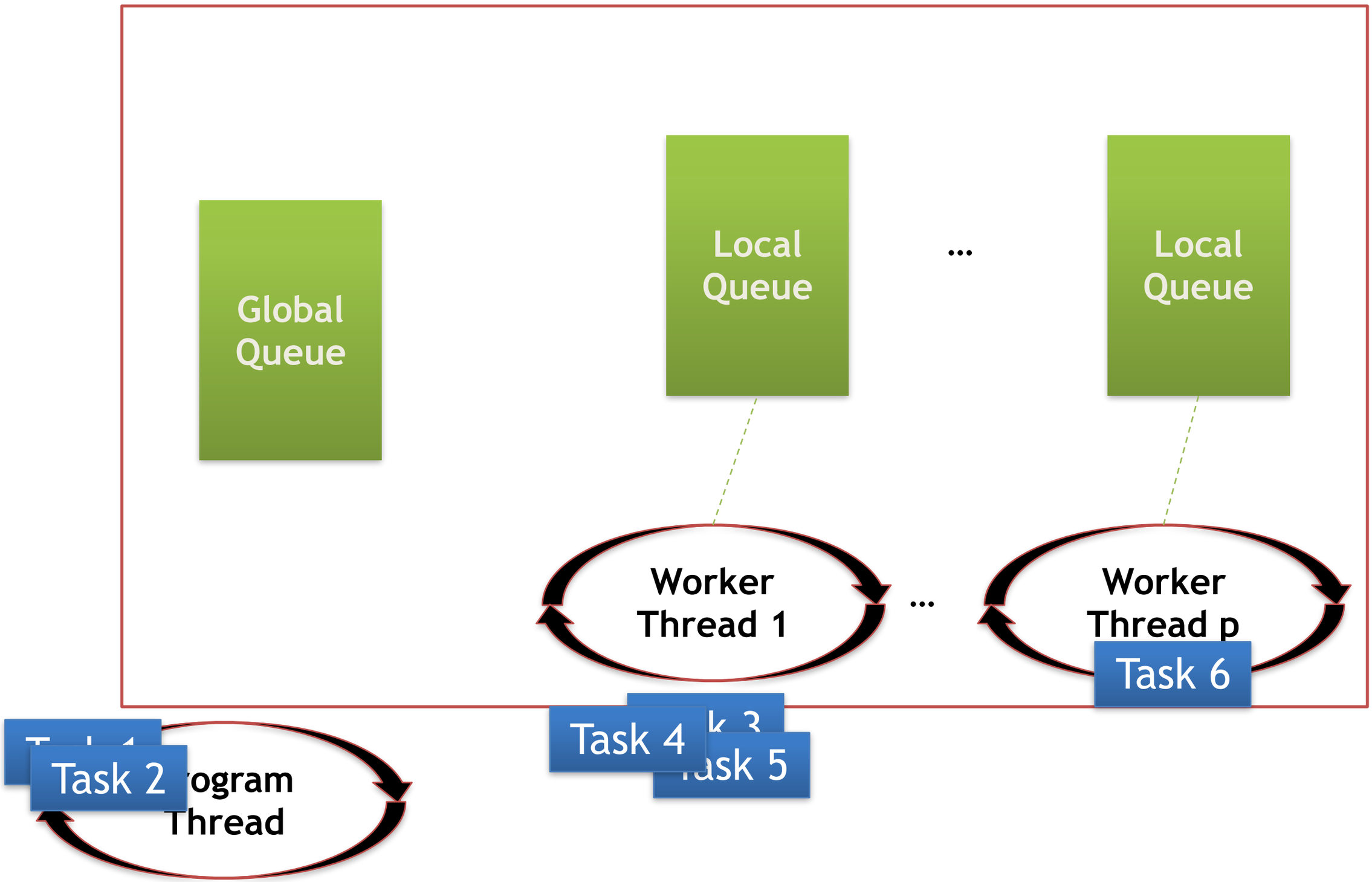
```
//for (int i = 0; i < 100; i++)
System.Threading.Parallel.For(0, 100,
        delegate(int i){  // parallel work here  } );
```

Parallel.ForEach

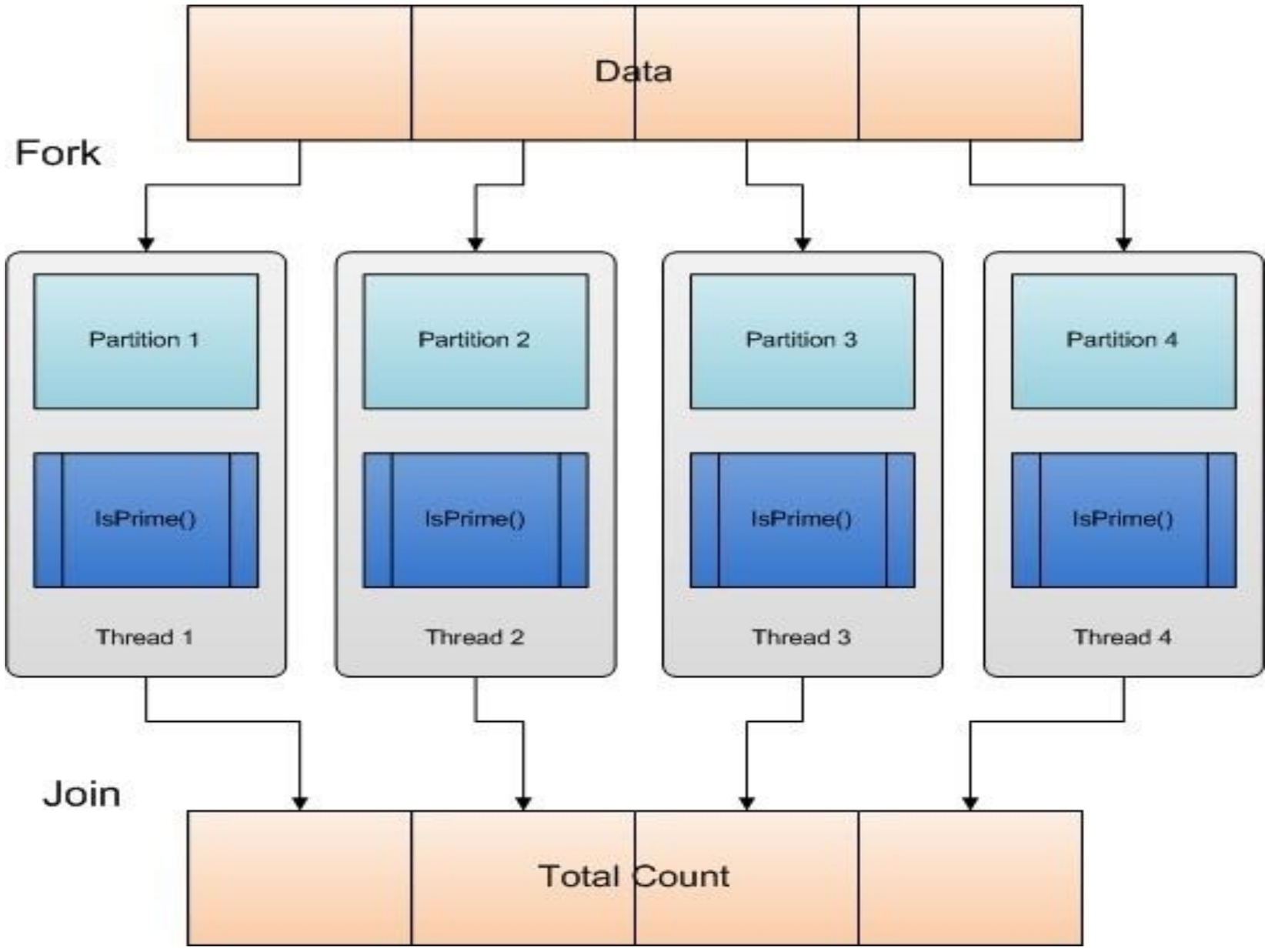
```
List<int> list = new List<int>();
Parallel.ForEach(list,
                item =>{  // parallel work here  });
```

Parallel.Invoke

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


Global Queue
Local Queue
...
Local Queue
Worker Thread 1
...
Worker Thread p
Task 6
Task 4
Task 5
Task 2
ogram Thread

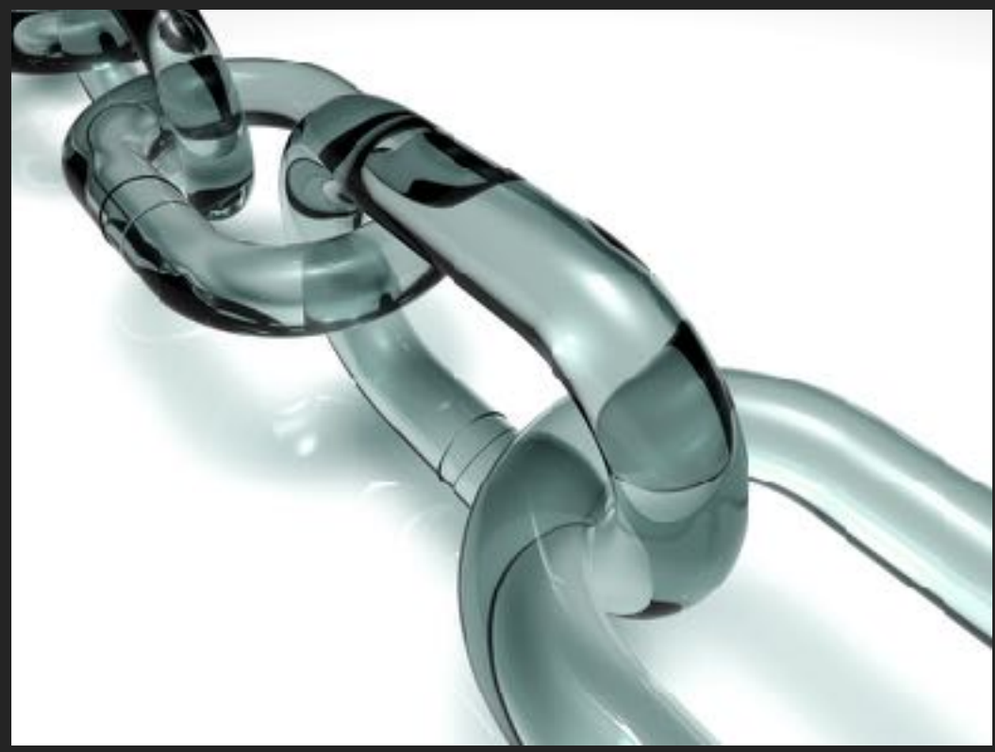
	Partitions work	Collates results
Parallel Stream	***Yes***	***Yes***
Thread class	No	No



10000 employees

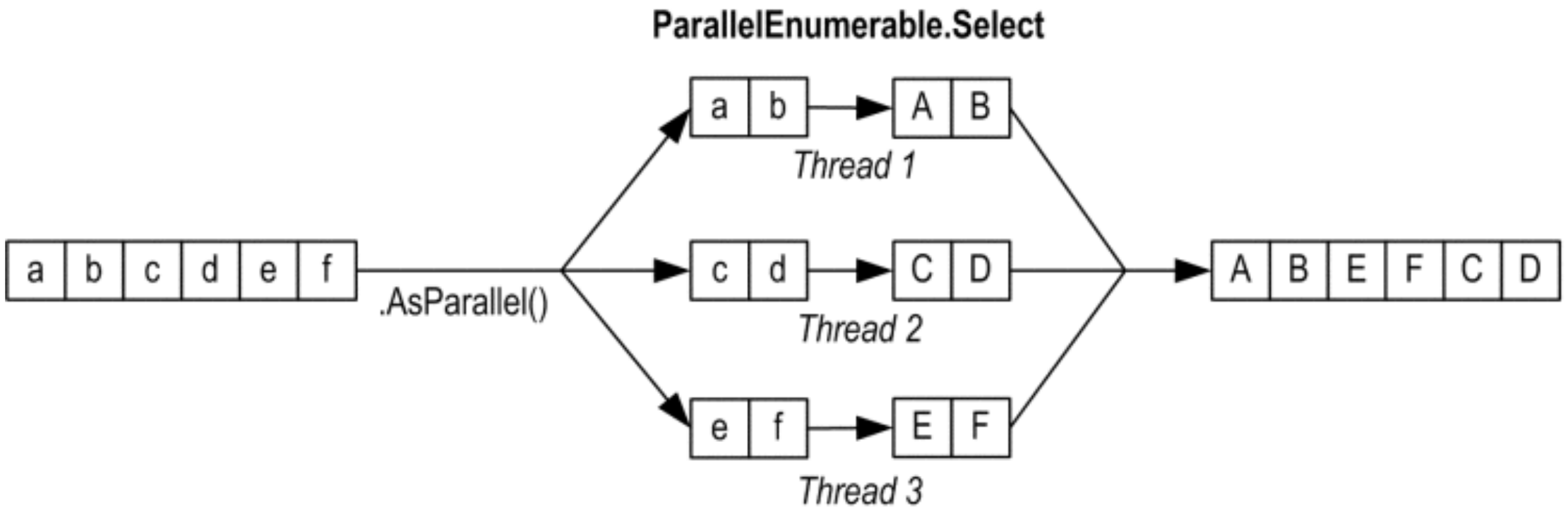
1 employee to each thread
100 employees to each thread
10000 employees in one thread

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()
```

Collection<10,000 employees>

#dedicated thread

```
for(Employee e: employees){
    Thread thread = new Thread(()-> save(e));
    Thread.start();
}
```

10,000 threads
Partition : 1 employee per thread

#threadpool

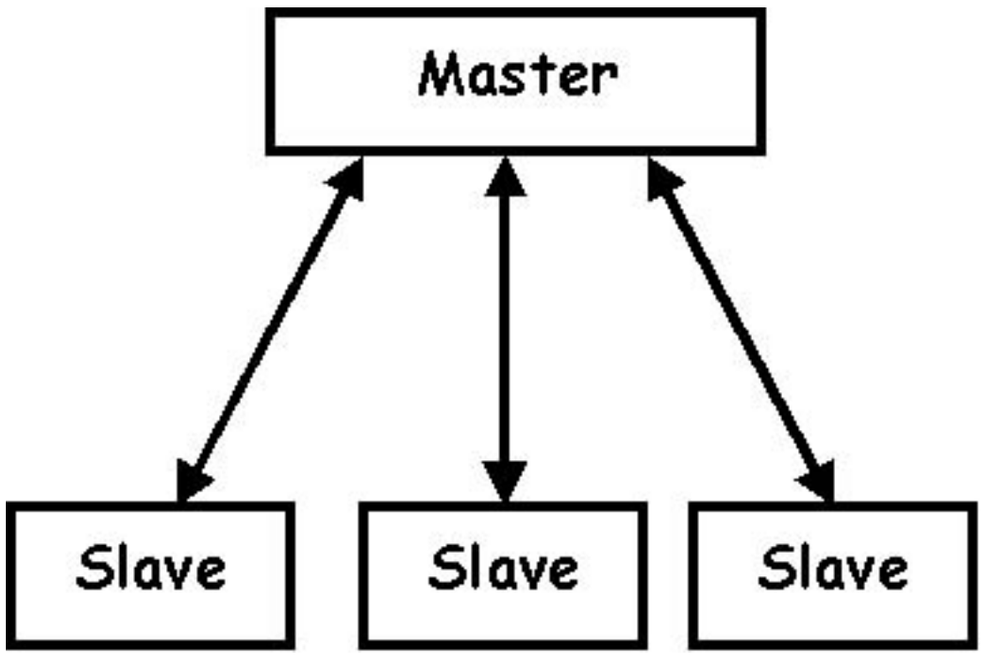
```
ExecutorService
executor=Executors.newFixedThreadPool(100);
for(Employee e: employees){
    executor.execute(()-> save(e));
}
```

100 threads
partition: 1 employees per thread
Reused 10,000 times

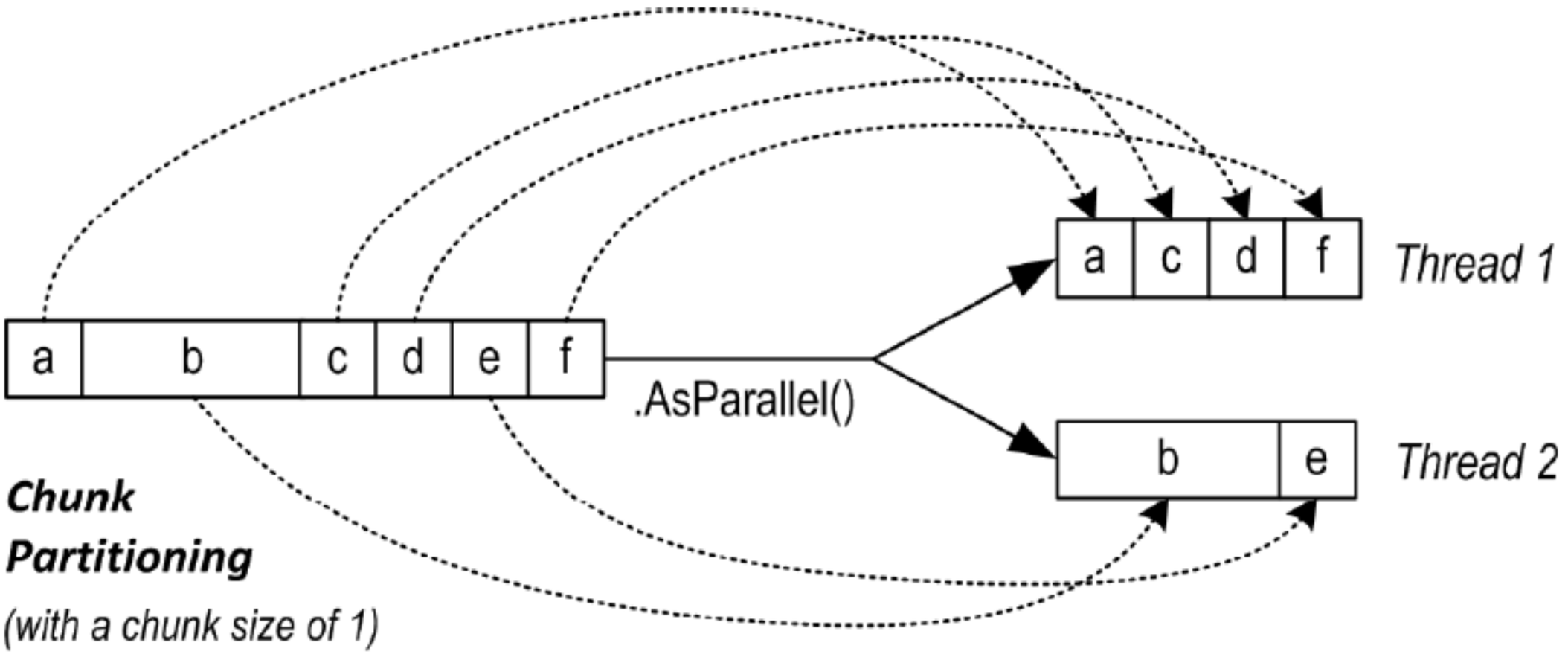
#task

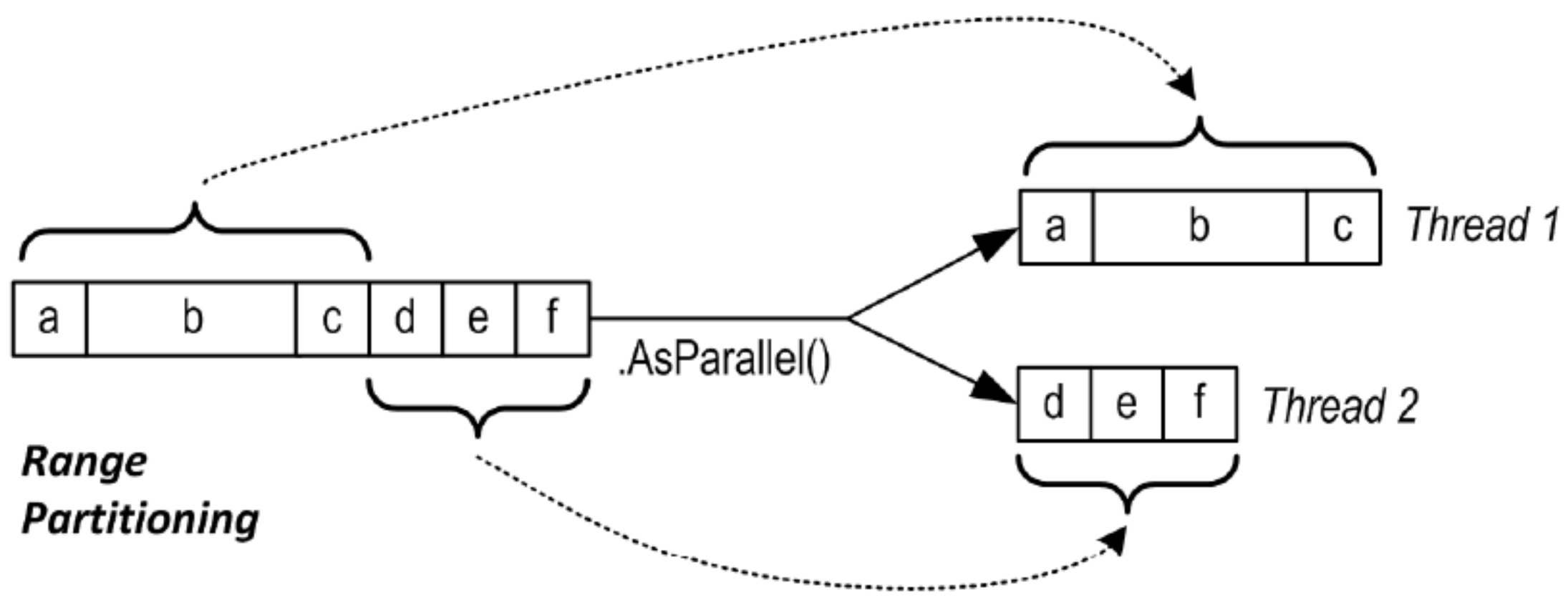
```
employees.parallelStream().forEach(Main::save);
```

100 threads
partition: 10 employees per thread
Reused 1000 times


Master
Slave
Slave
Slave

Why Isn’t .AsParallel() the Default?


a
b
c
d
e
f
.AsParallel()
a
c
d
f
Thread 1
b
e
Thread 2
Chunk
Partitioning
(with a chunk size of 1)


a
b
c
d
e
f
.AsParallel()
a
b
c
Thread 1
d
e
f
Thread 2
Range
Partitioning

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**
Chunk	***Dynamic***	***Average***
Range	Static	Poor to excellent
Hash	***Static***	***Poor***
Stripe	Static	Poor to excellent
Custom	***Dynamic/static***	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).

How Windows does Synchronous I/O

.NET

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

① ⑨

Win32 User-Mode

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

③ IRP ② ⑧

Windows Kernel-Mode

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

④ ⑦

Your thread blocks here!
Hardward does I/O; ⑥
No threads involved!

NTFS Driver

IRP Queue

⑤

Network

DVD-ROM

RS-232

How Windows does Asynchronous I/O

.NET

```
FileStream fs = new FileStream(..., FileOptions.Asynchronous);
fs.BeginRead(..., CallbackMethod, ...);
```

```
void CallbackMethod(...) { ... }
```

Win32 User-Mode

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

IRP

Windows Kernel-Mode

(Windows I/O Subsystem Dispatcher code)

1 2 3 4 5 6 7 a b c

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

NTFS Driver

IRP Queue

The CLR's Thread Pool

Threads can extract completed IRP's from here

File.read()

Post processing

paysalary()

new thread(read)

paysalary()

File.readAsync(callback)

paysalary()

File.read()

Post processing

callback()

Post processing

Client1.ConnectAsync()

Client2.ConnectAsync()

Client3.ConnectAsync()

paysalary()

callback()

Post processing

Selector

callback()

Post processing

callback()

Post processing

- a + b -> 3 cpu cycles
- Fun call -> 10 cpu cycles
- Create thread -> 200,000
- Destroy thread -> 100,000
- Switch context - > 4000
- Write file -> 10,00,000
-

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

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

Sync IO

suspend the calling thread while hardware performs the work.

Non-blocking Async IO

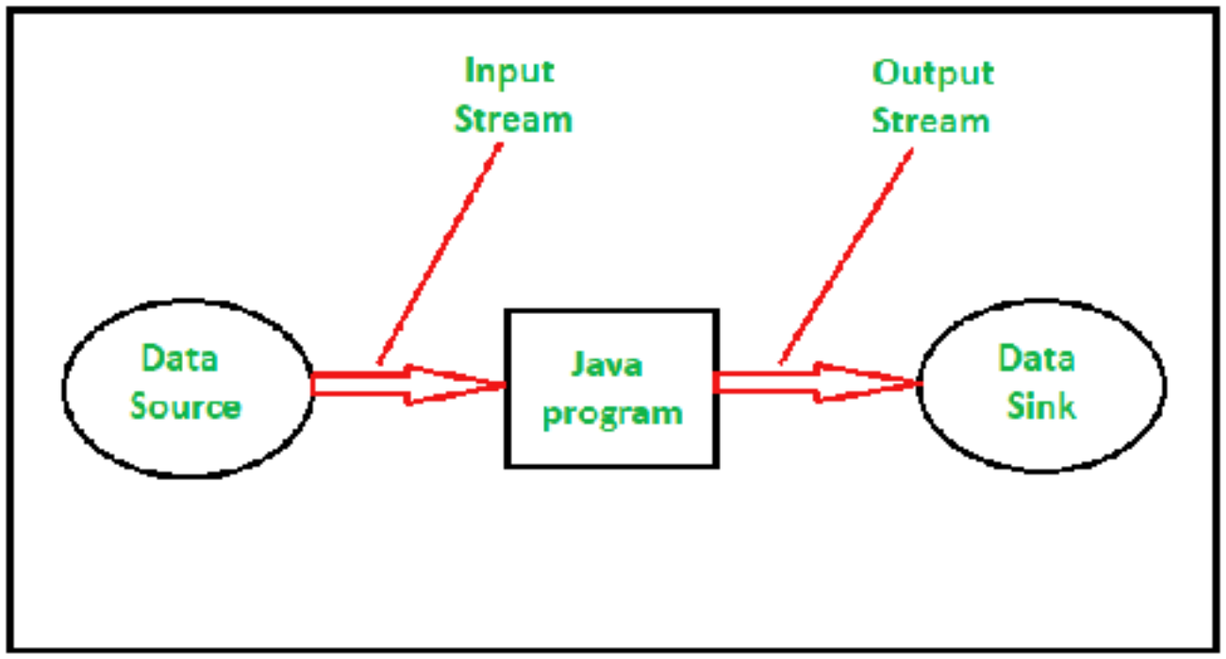
The kernel doesn't block your thread and returns from the IO call immediately. YYour code then needs to check occasionally if the kernel has already done its job. As an example, Linux provides the epoll interface for the non-blocking IO.

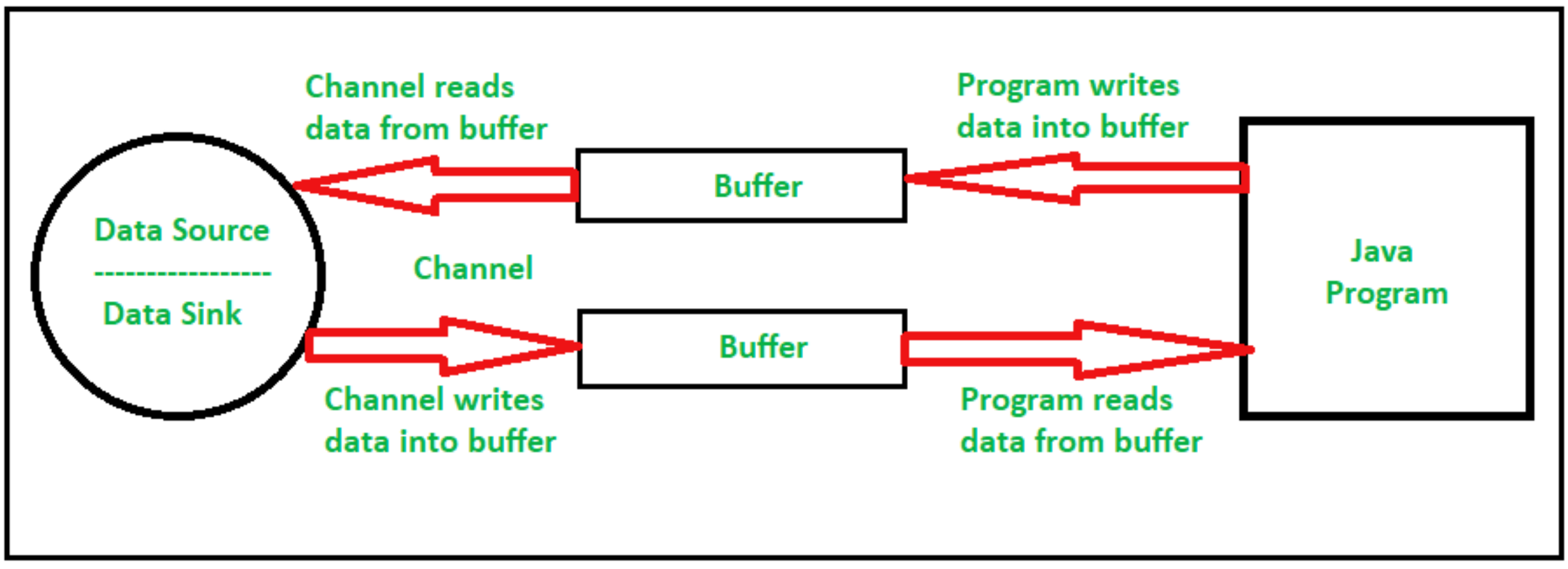
Blocking Async IO

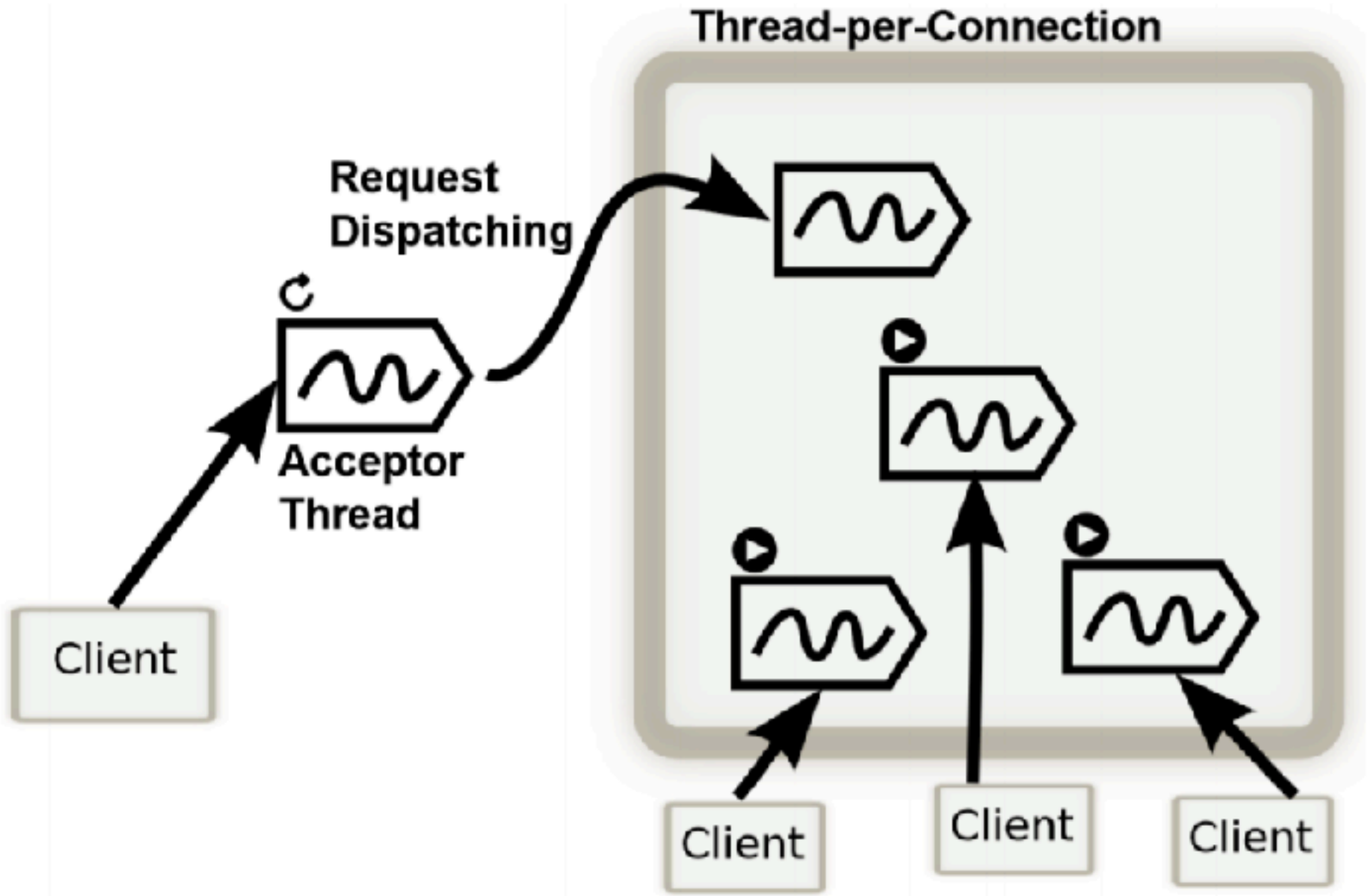
There is still a thread lower down the stack which is blocked on the IO, but your thread isn't.
called wrapped inside a thread which will block until the IO is done and then delegate the handling of the result of the IO to your callback function.

hardware (disk-drive, network interface card, etc.)

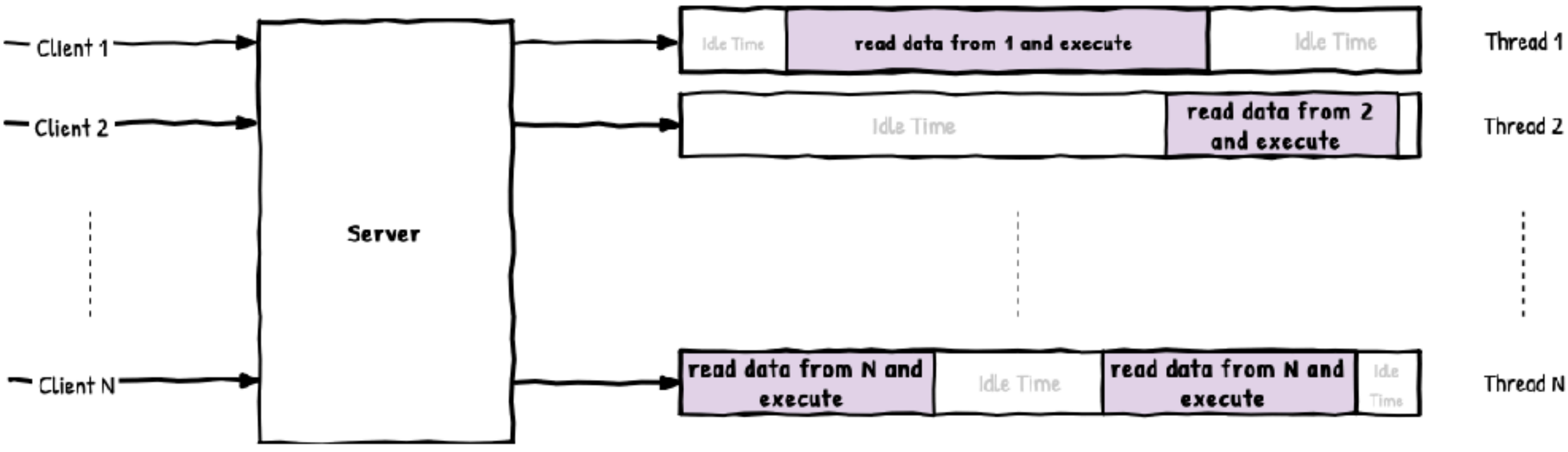
Blocking vs. Non-blocking IO

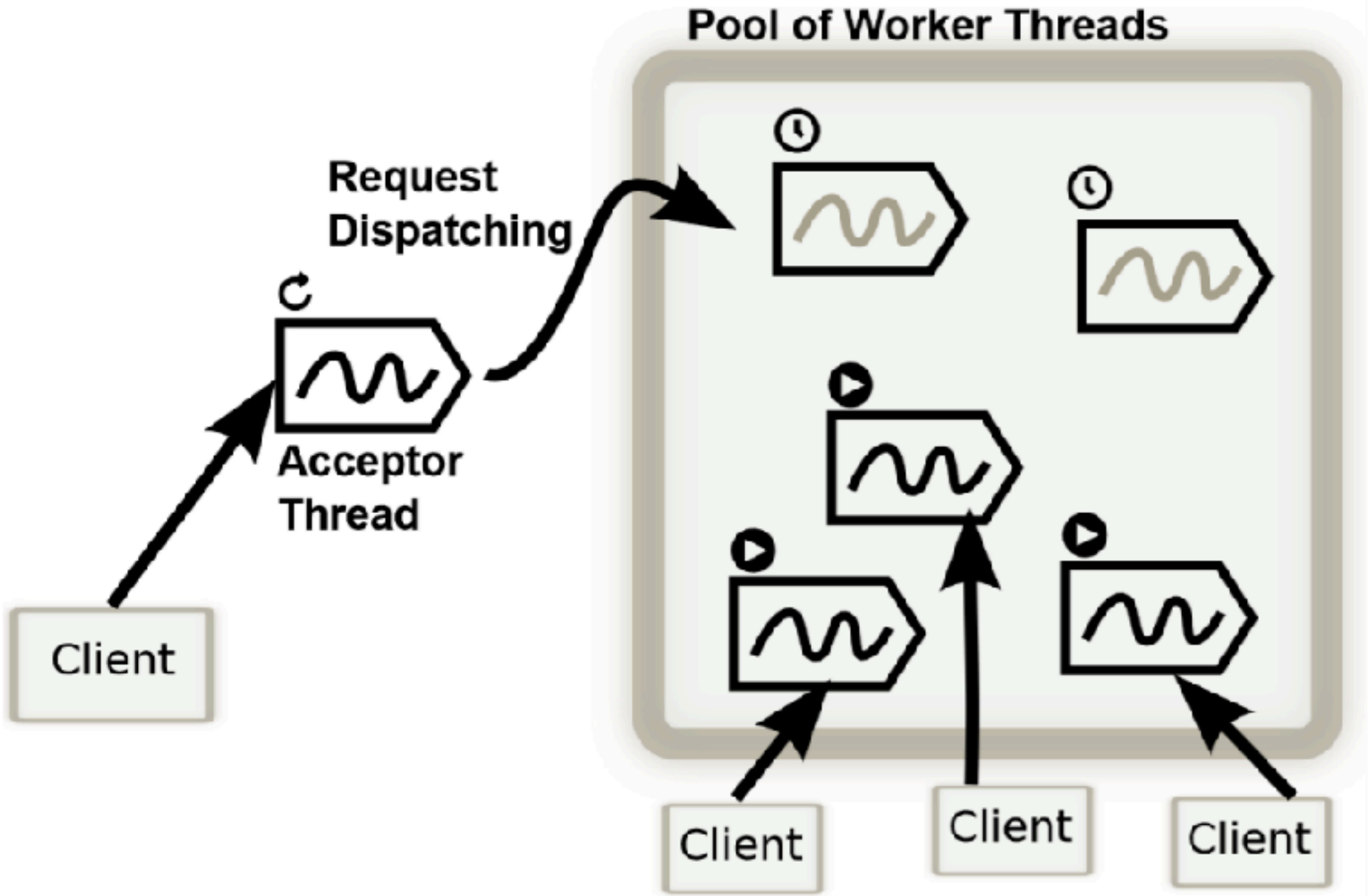


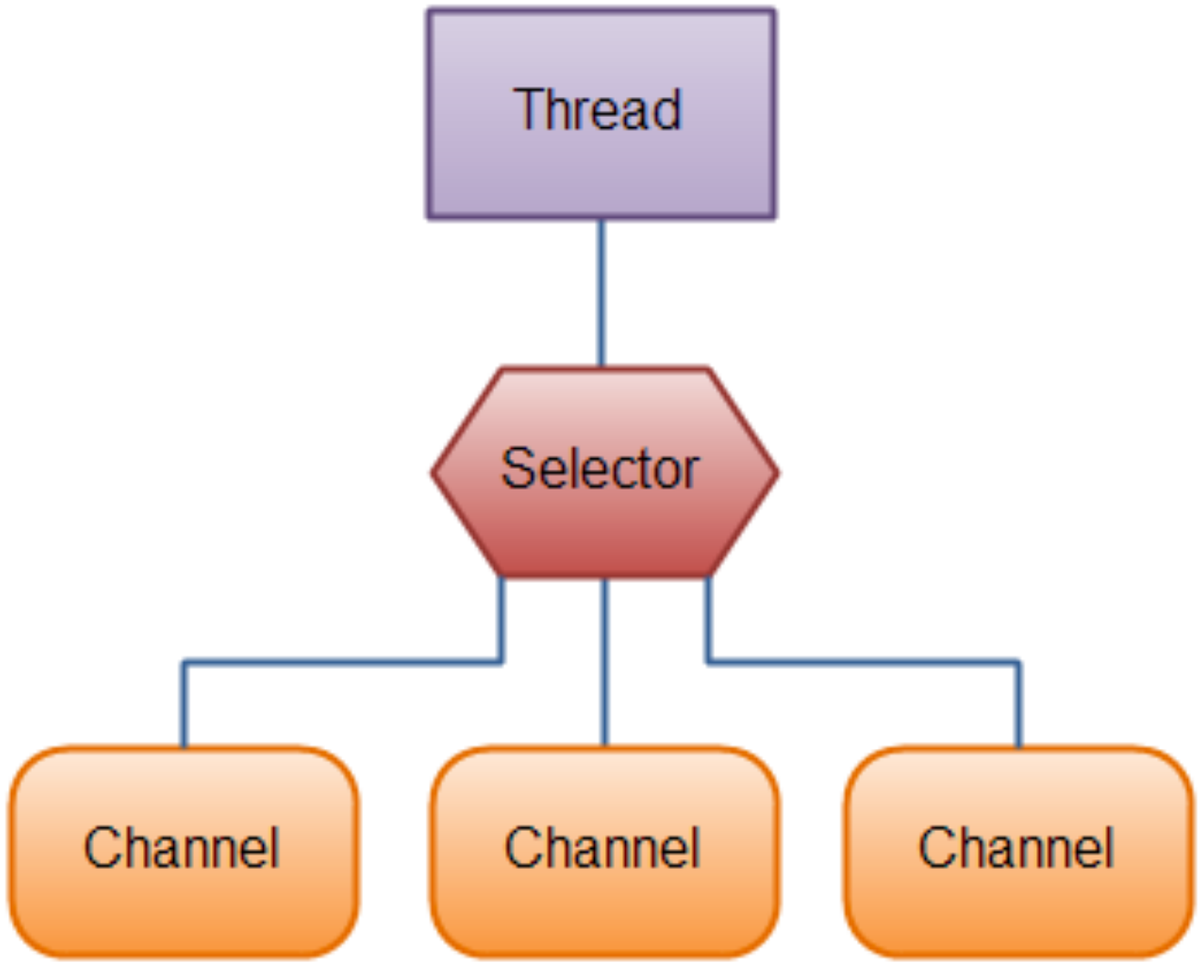



Thread-per-Connection
Request
Dispatching
Acceptor
Thread
Client
Client
Client
Client

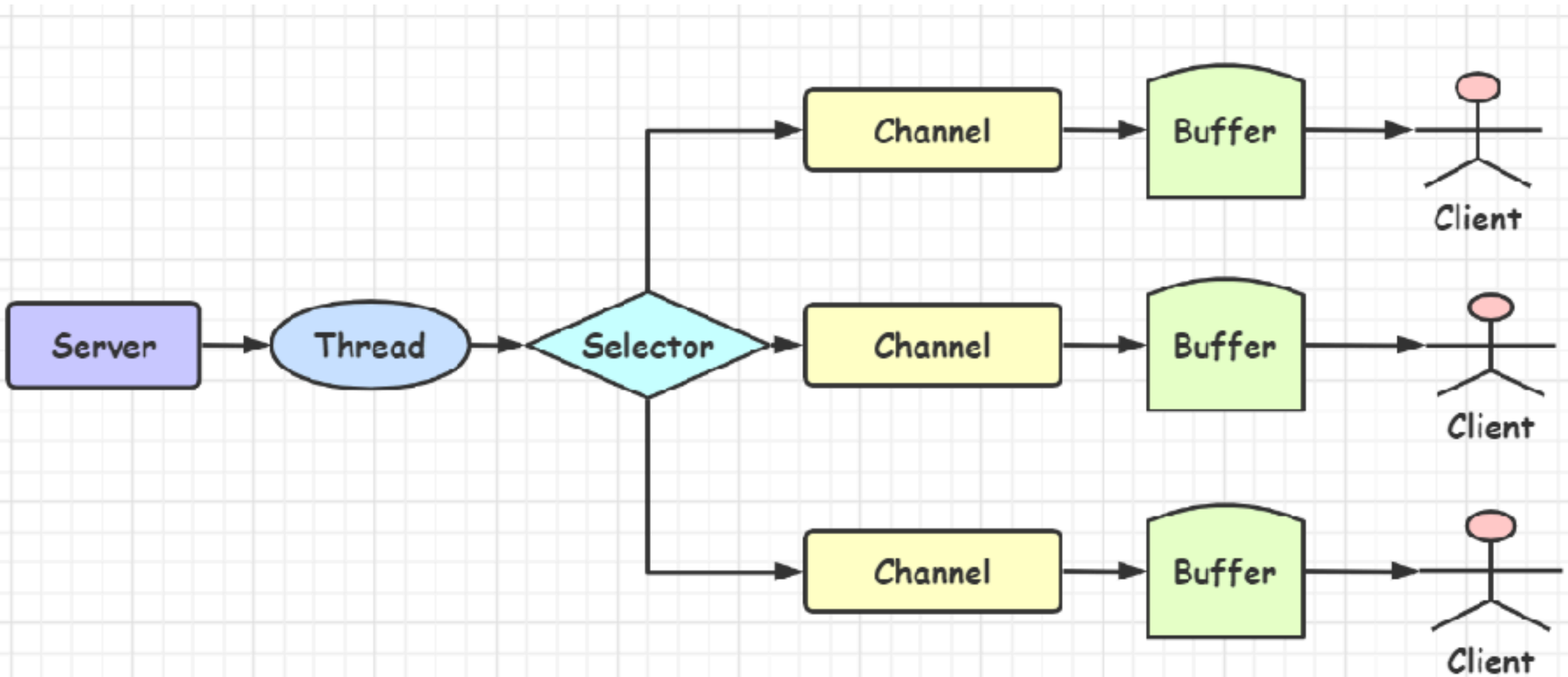
Server with a thread per connection (Blocking sockets)

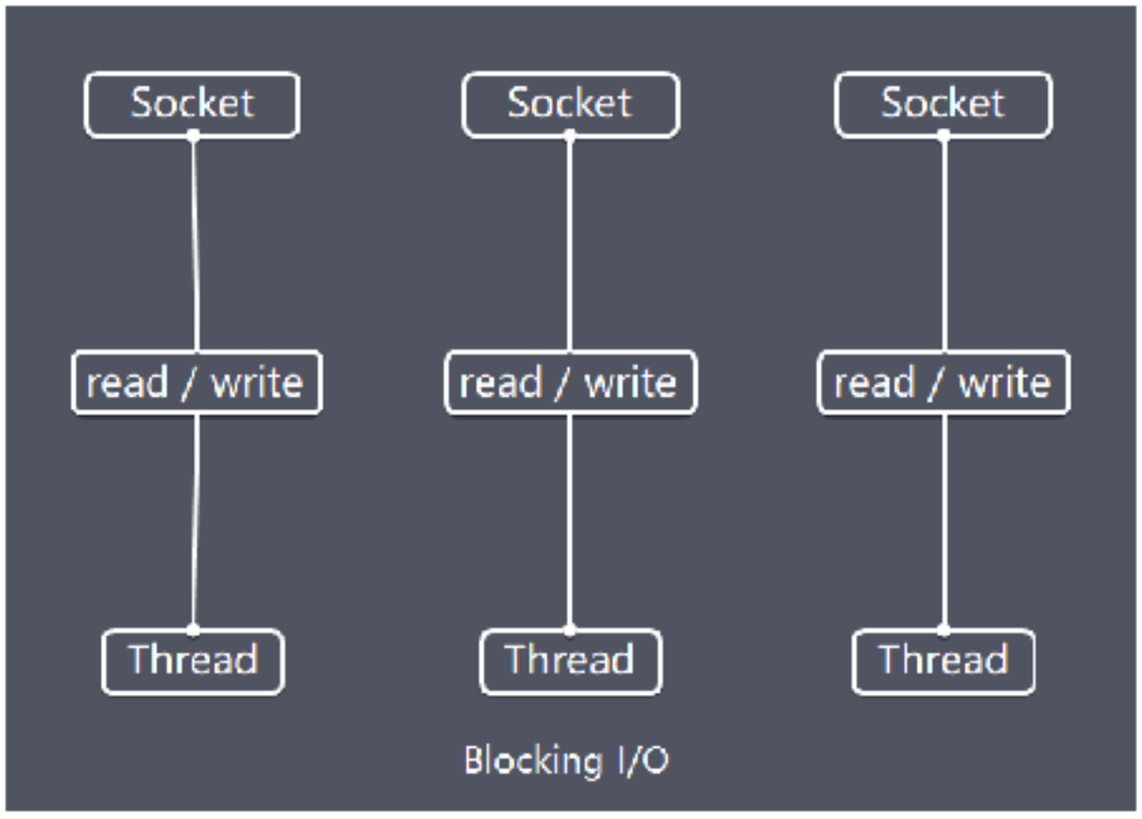


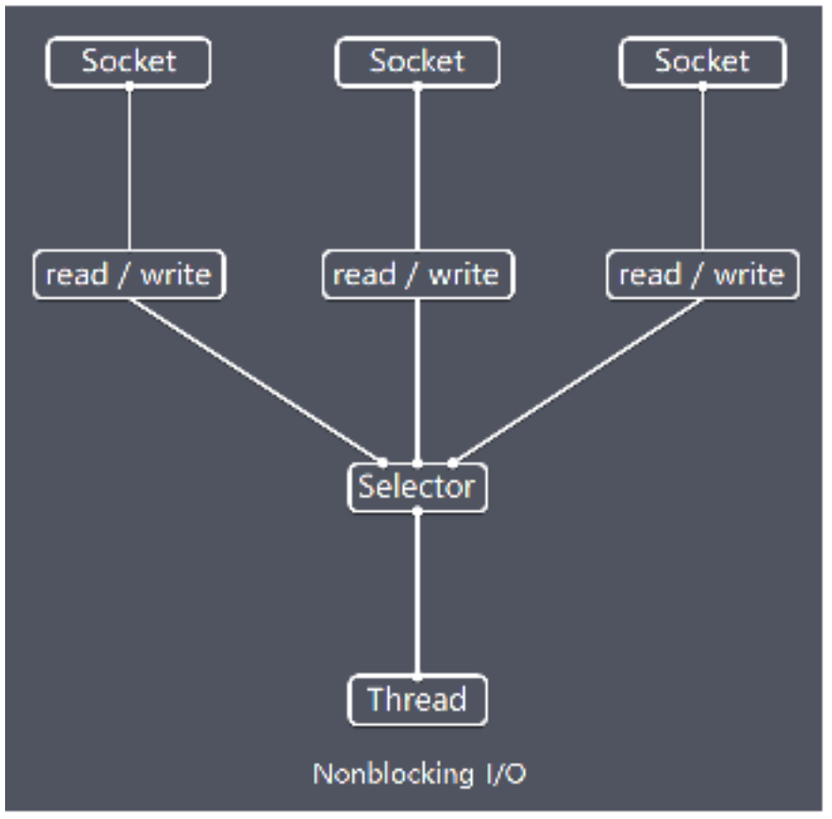

Pool of Worker Threads
Request
Dispatching
Acceptor
Thread
Client
Client
Client
Client

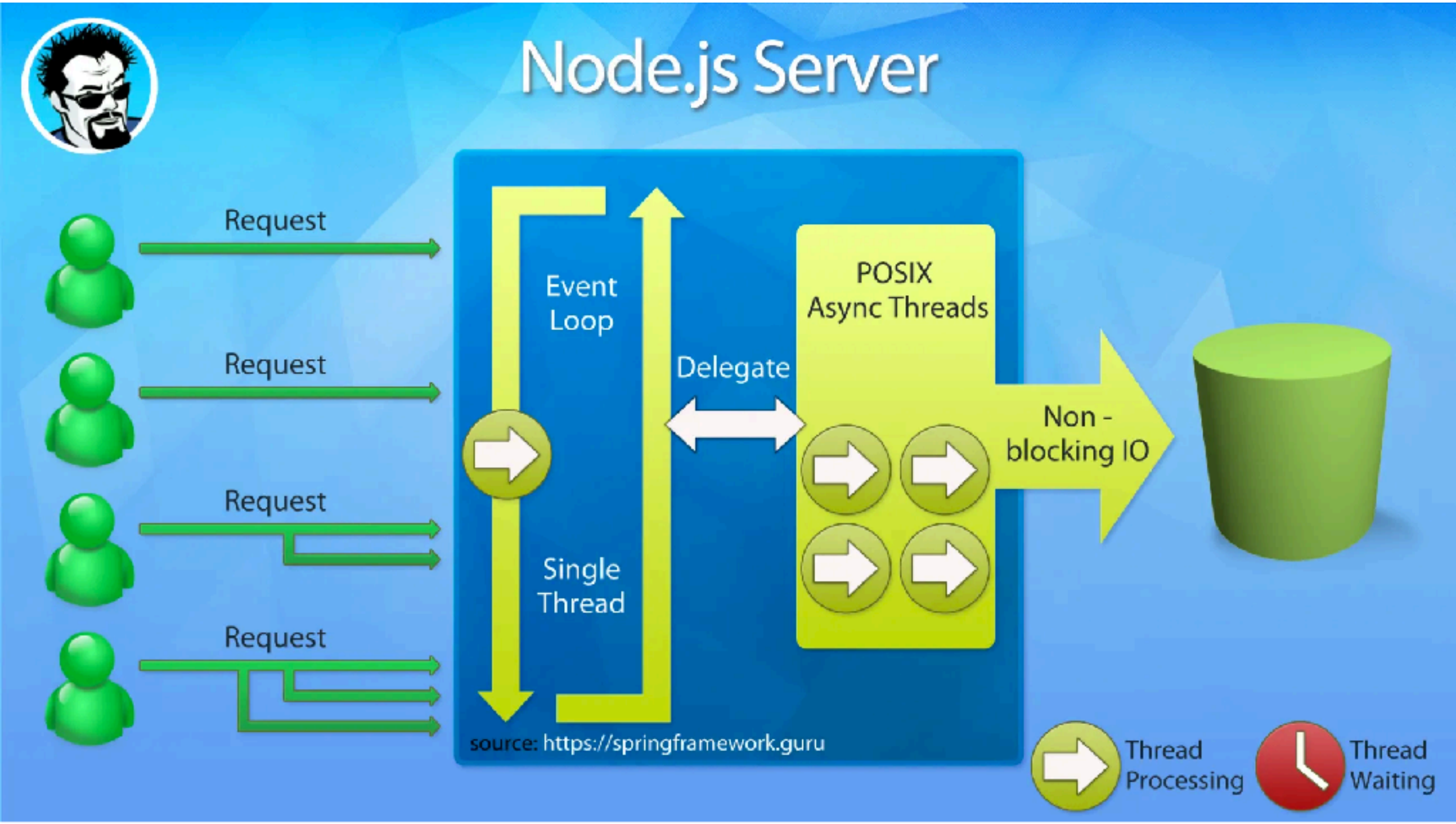


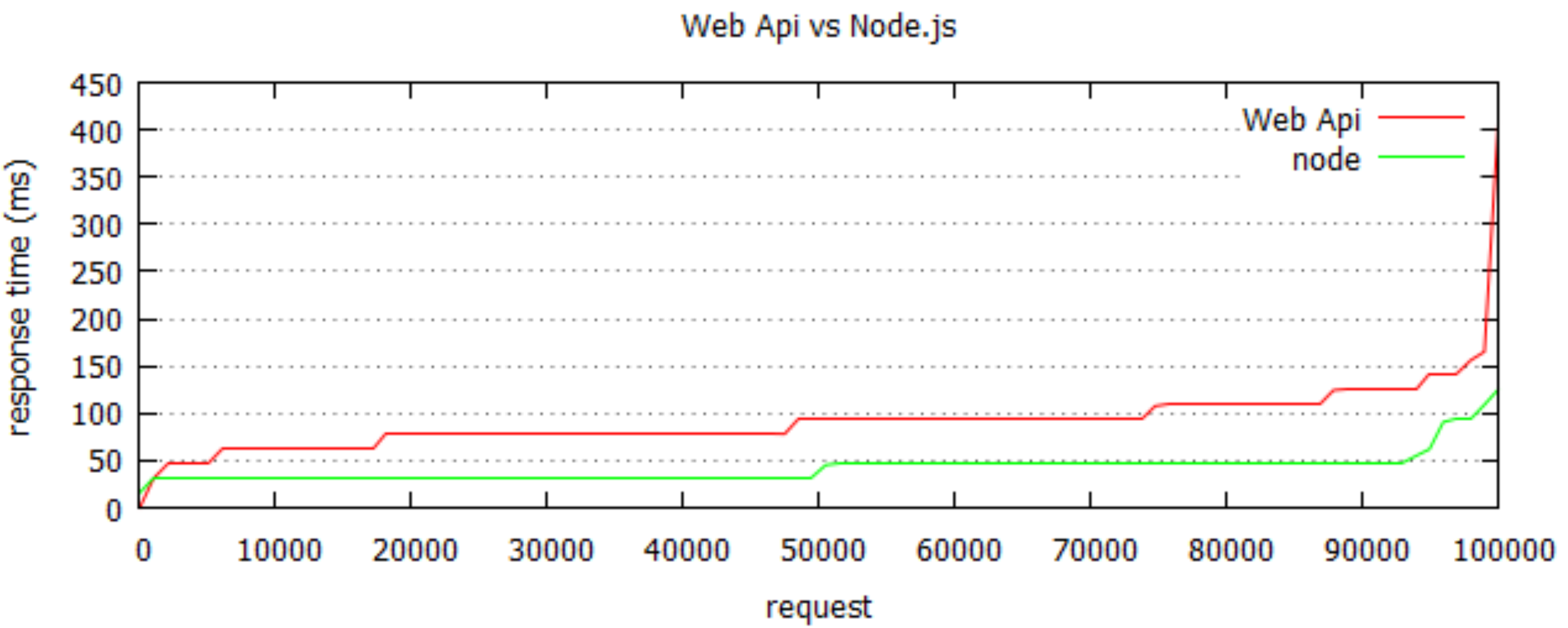
one thread to handle all of your channels. Switching between threads is expensive for an operating system, and each thread takes up some resources (memory) in the operating system too. Therefore, the less threads you use, the better.

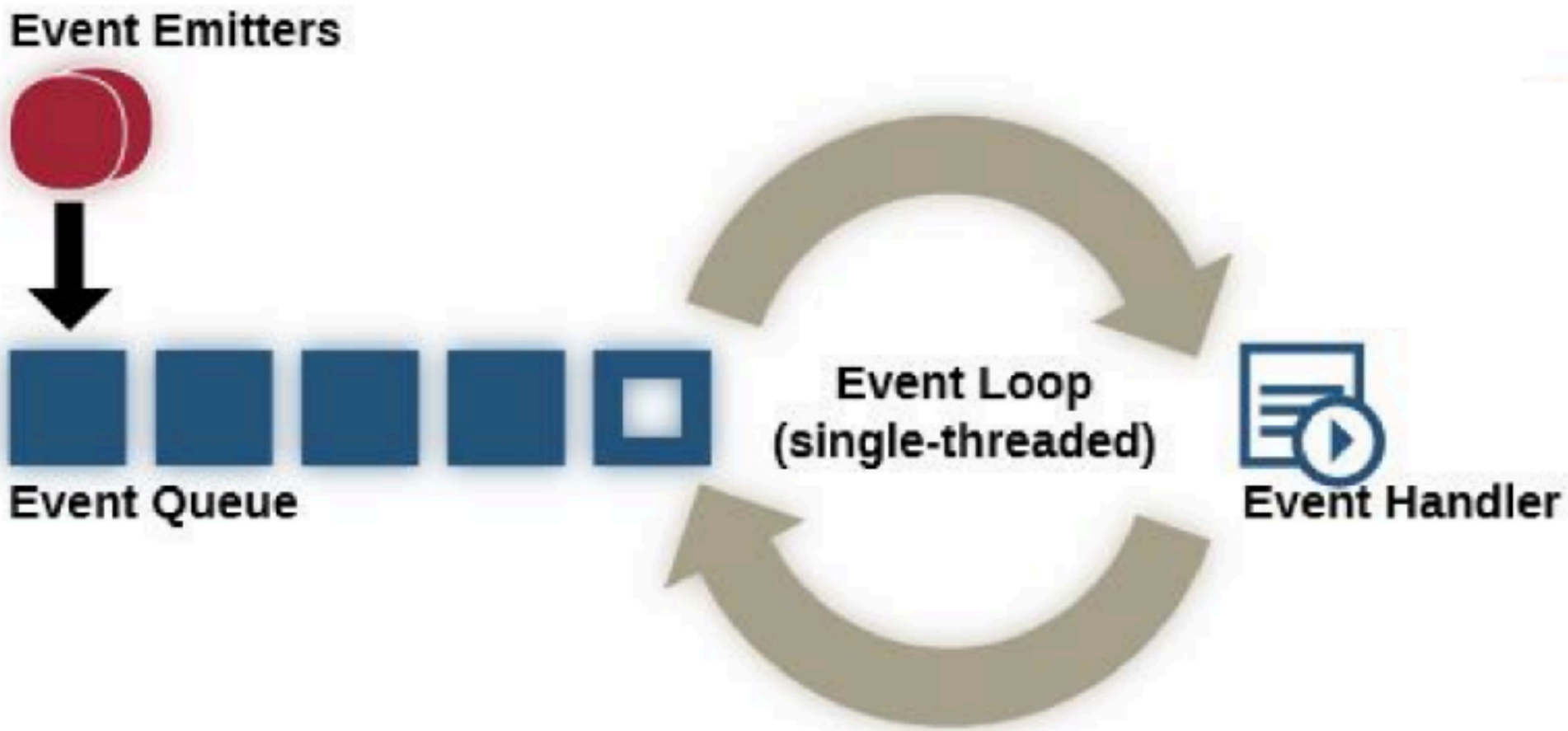

Server
Thread
Selector
Channel
Buffer
Client
Channel
Buffer
Client
Channel
Buffer
Client


Socket
Socket
Socket
read / write
read / write
read / write
Thread
Thread
Thread
Blocking I/O

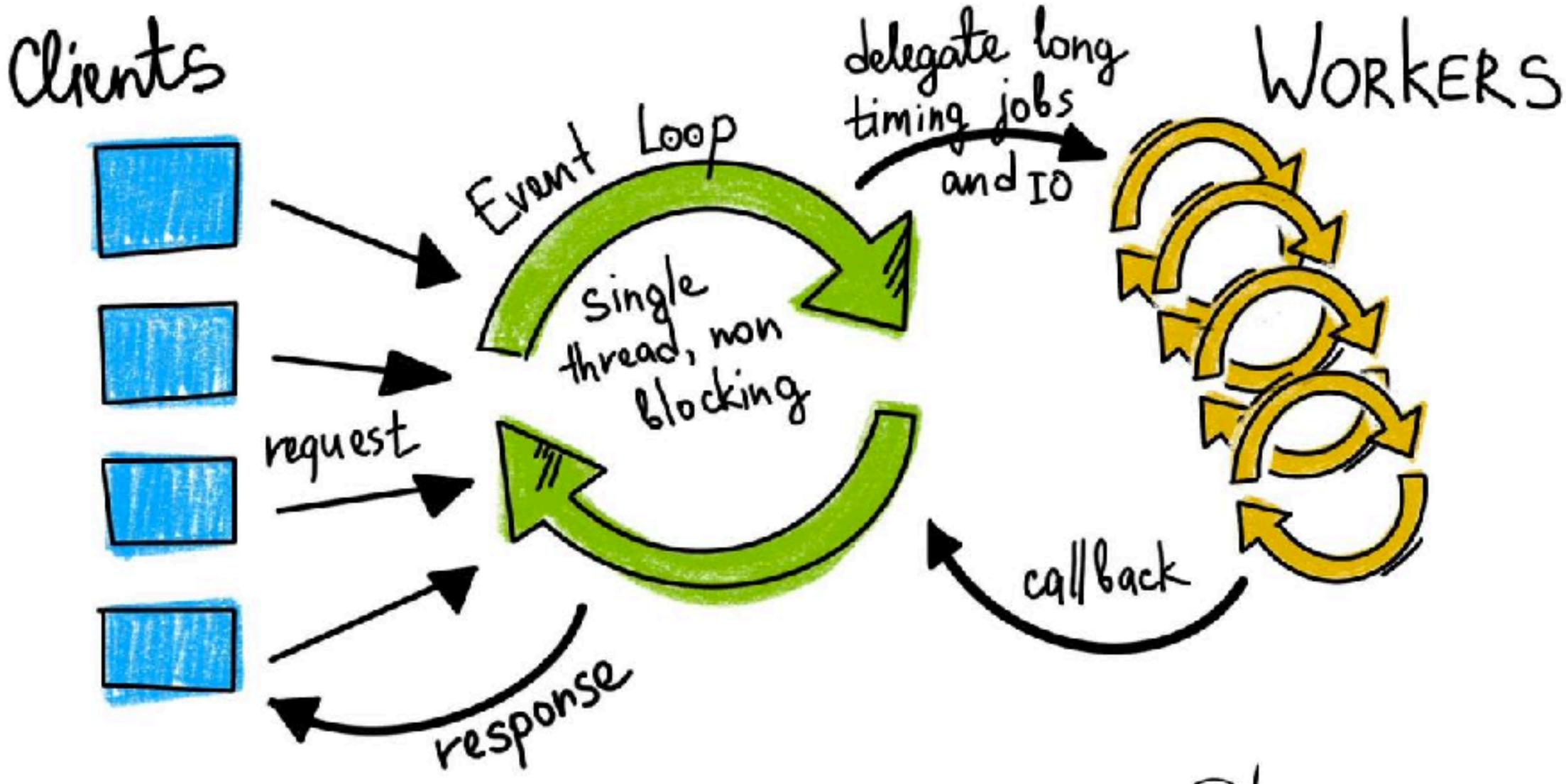

Socket
Socket
Socket
read / write
read / write
read / write
Selector
Thread
Nonblocking I/O


Node.js Server
Request
Request
Request
Request
Event Loop
Delegate
POSIX Async Threads
Non - blocking IO
Single Thread
source: https://springframework.guru
Thread Processing
Thread Waiting

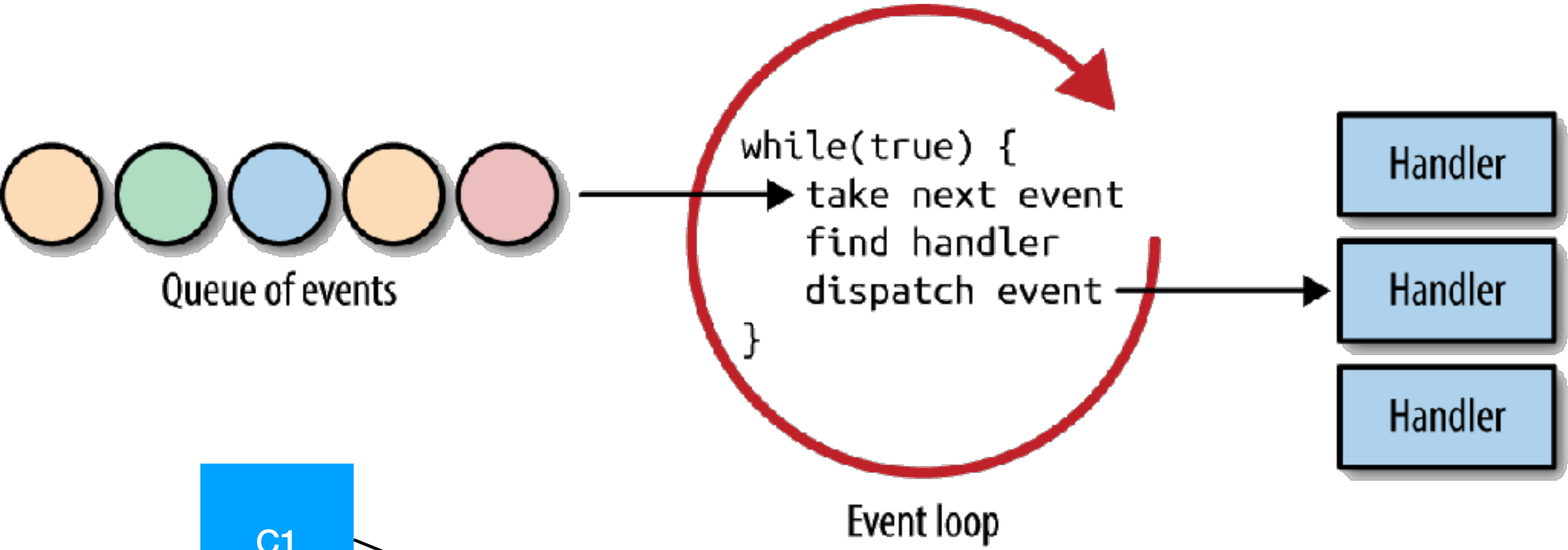

Web Api vs Node.js
response time (ms)
450
400
350
300
250
200
150
100
50
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
request
Web Api
node

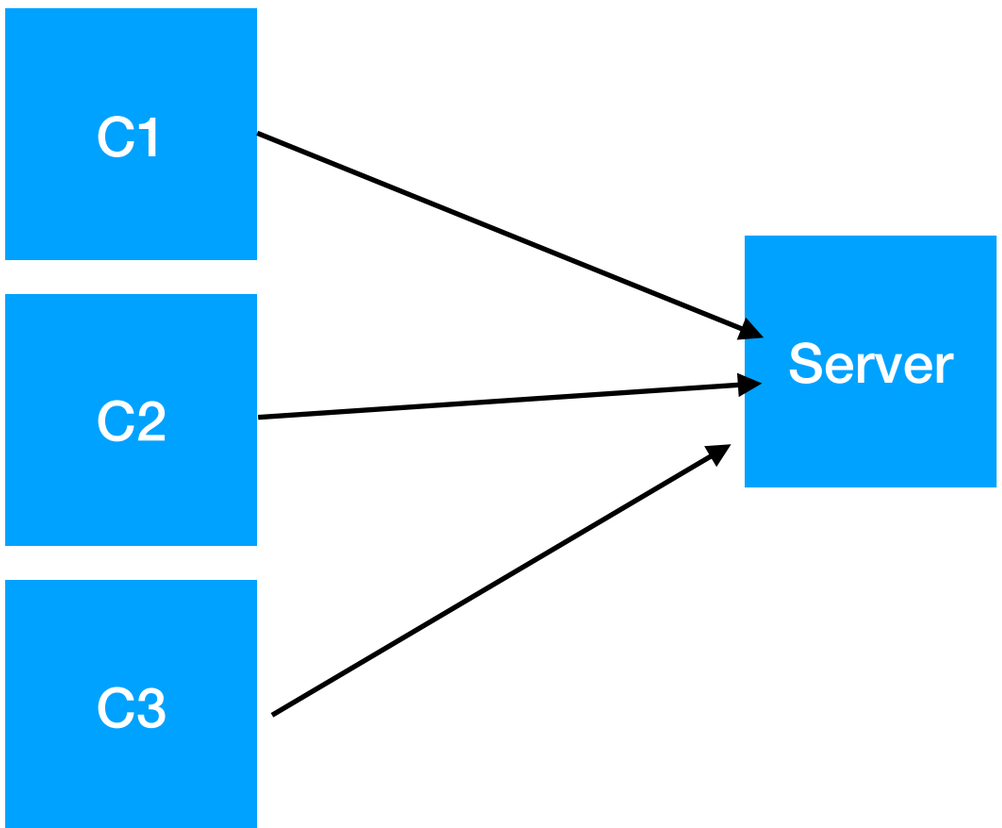


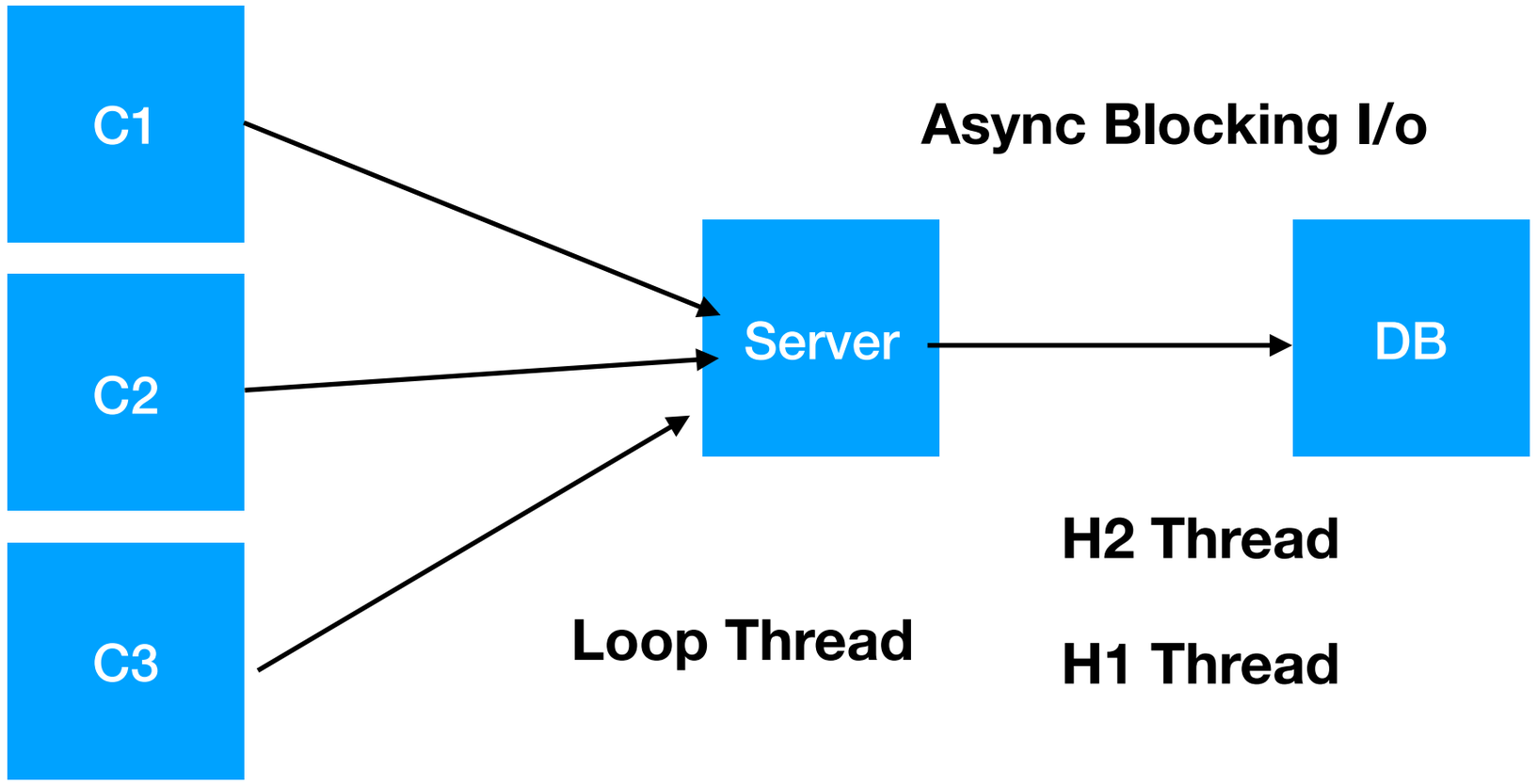
The event demultiplexor delivers the requested events to the event handlers.

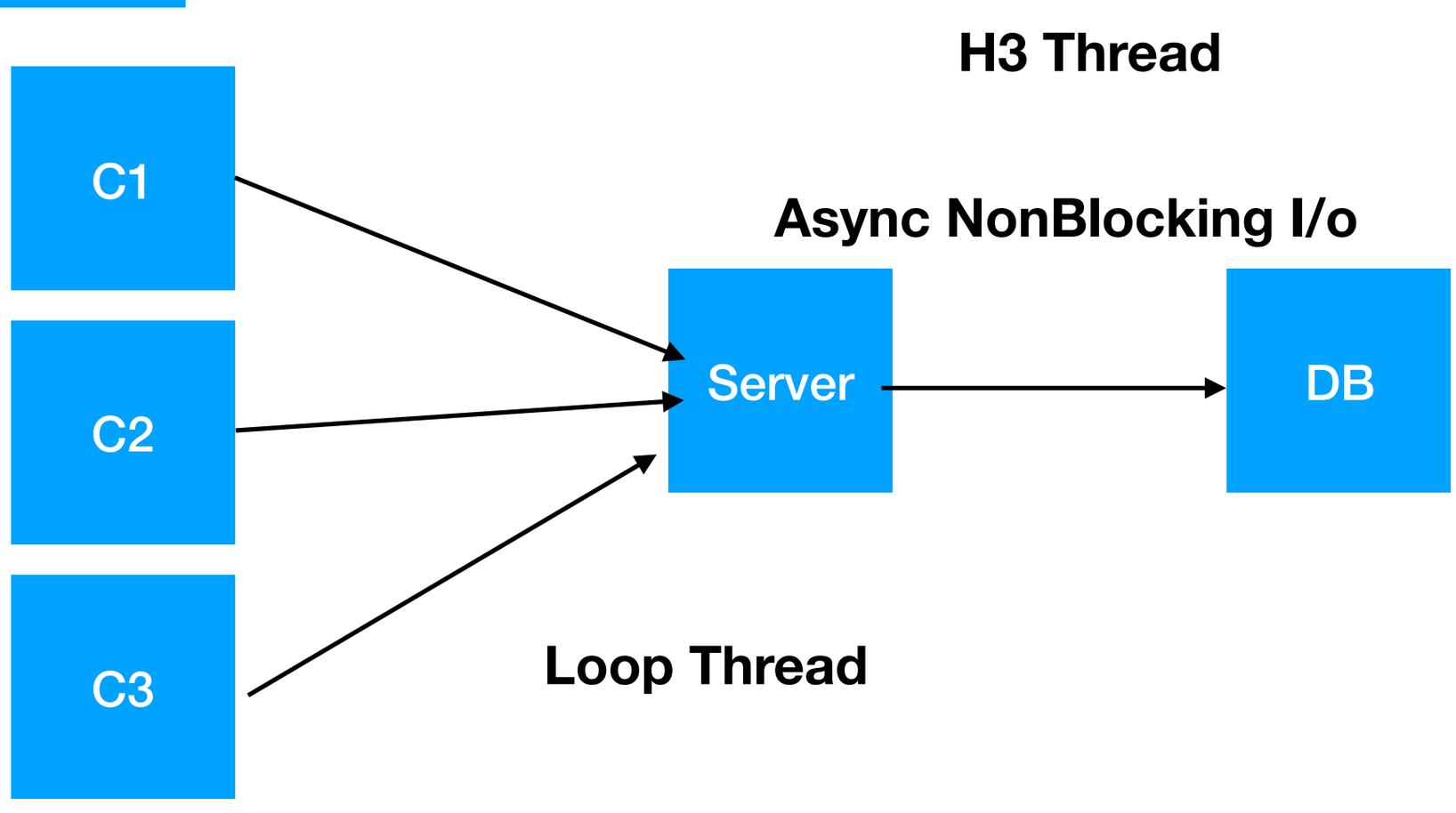


@luminousmen.com

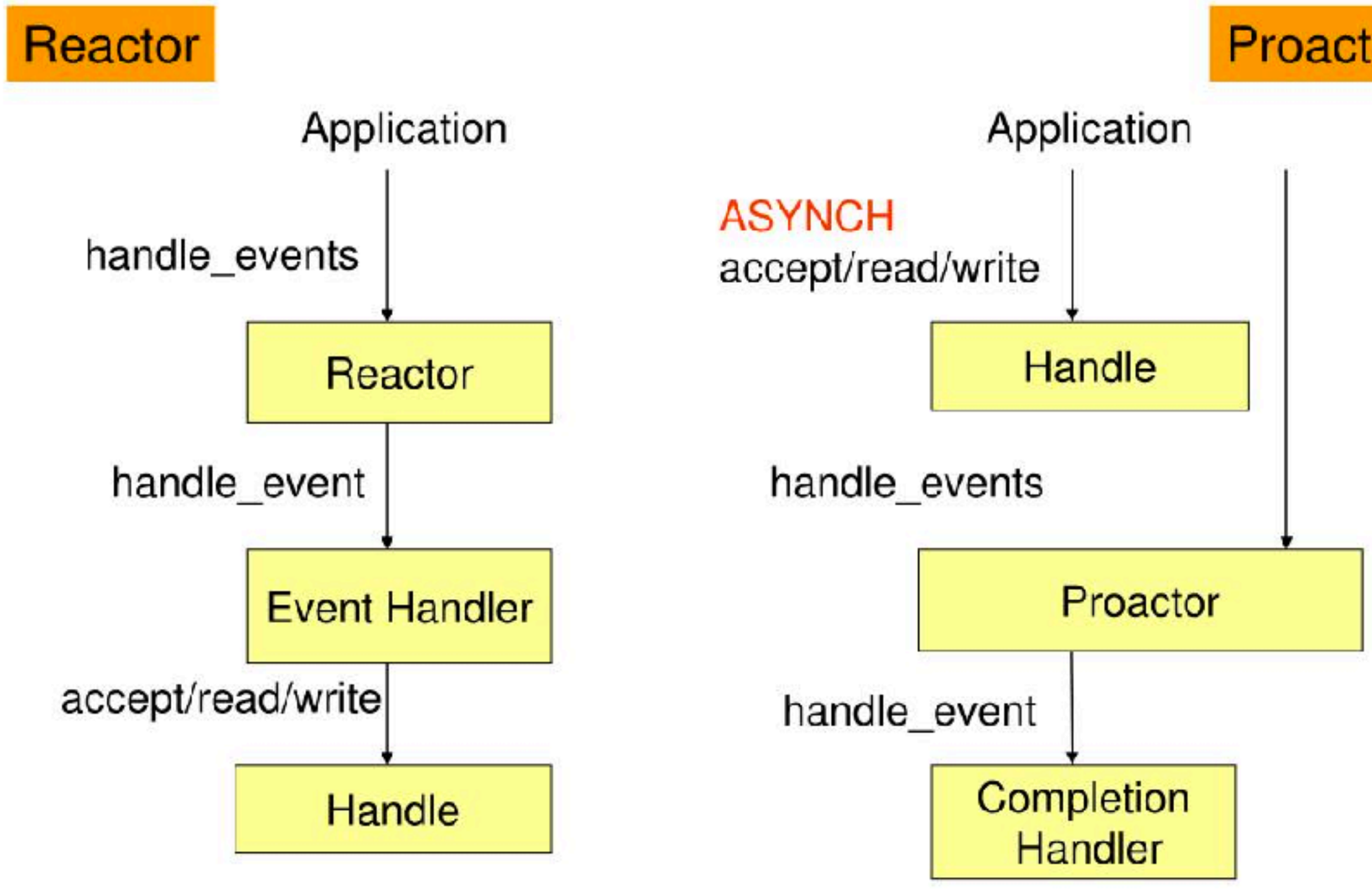

while(true) {
take next event
find handler
dispatch event
}
Queue of events
Handler
Handler
Handler
Event loop


C1
C2
C3
Server

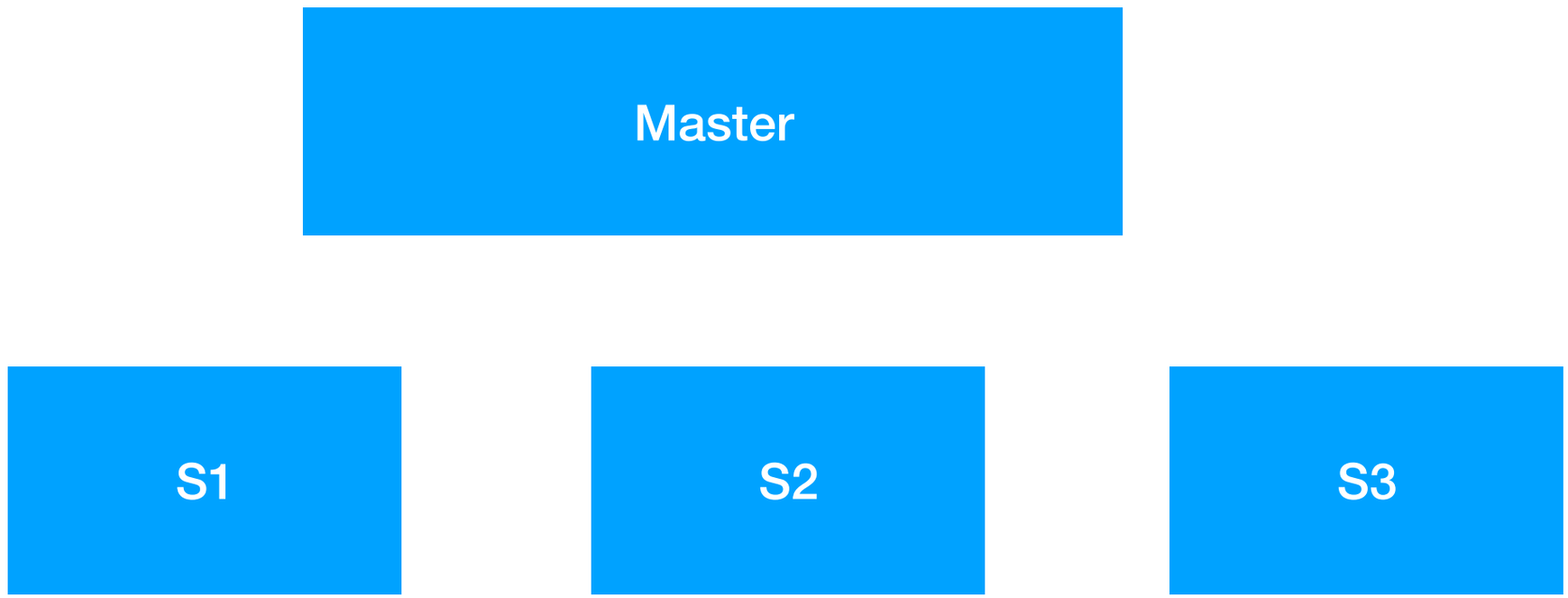

C1
C2
C3
Server
DB
Async Blocking I/o
Loop Thread
H2 Thread
H1 Thread


H3 Thread
C1
C2
C3
Server
DB
Async NonBlocking I/o
Loop Thread

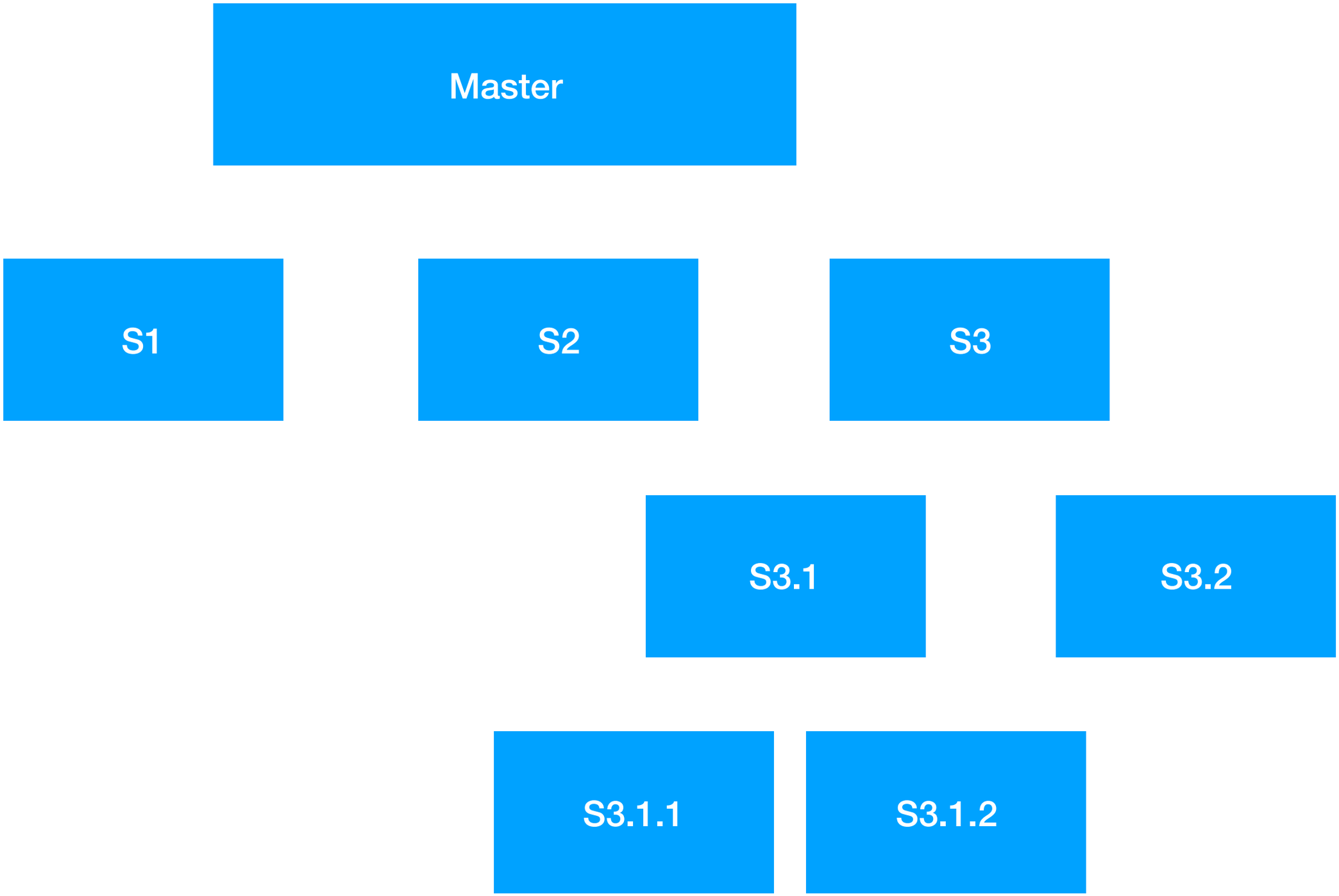
Compare Reactor vs. Proactor Side by Side

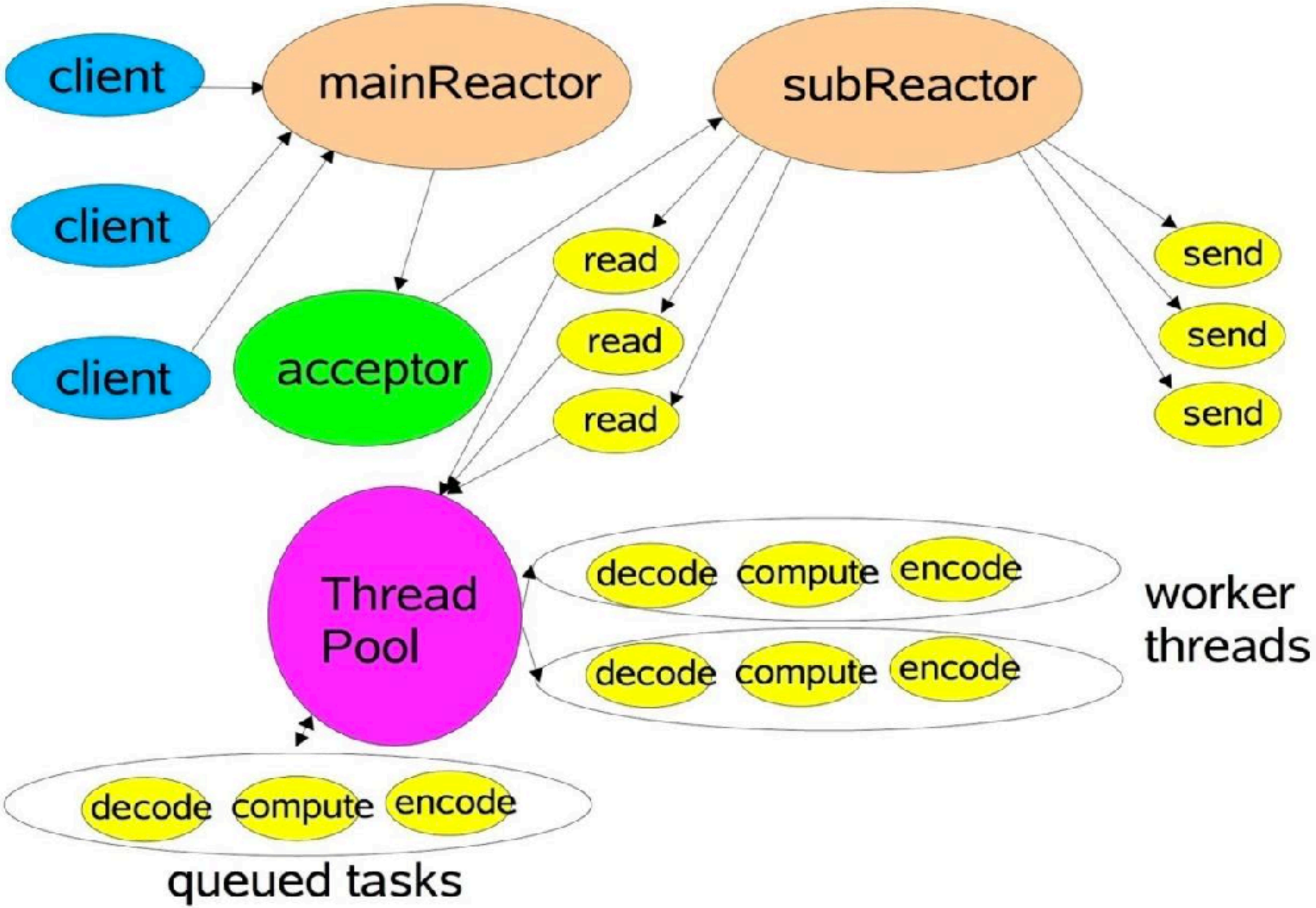


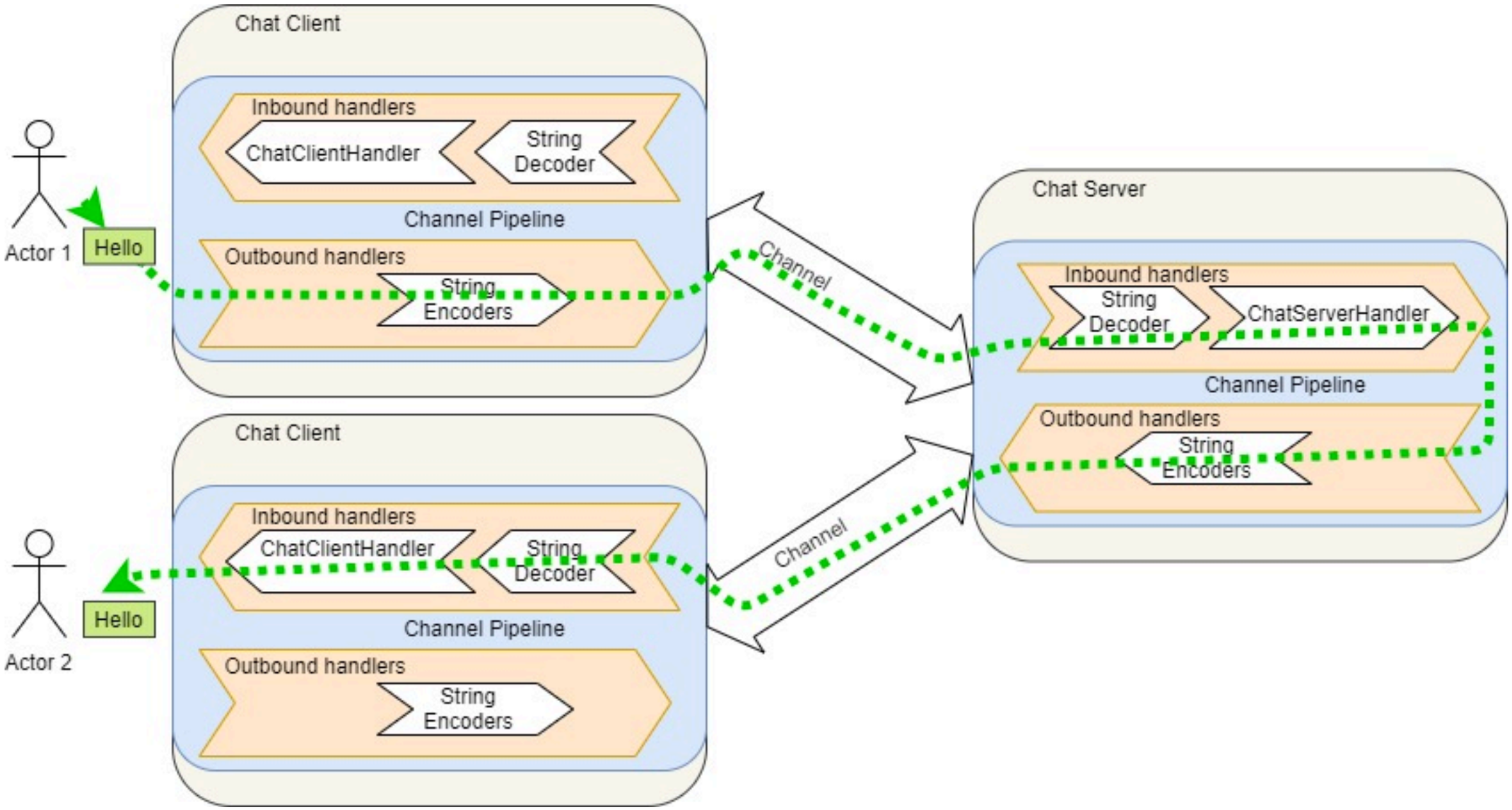
Master Slave



Fork and Join

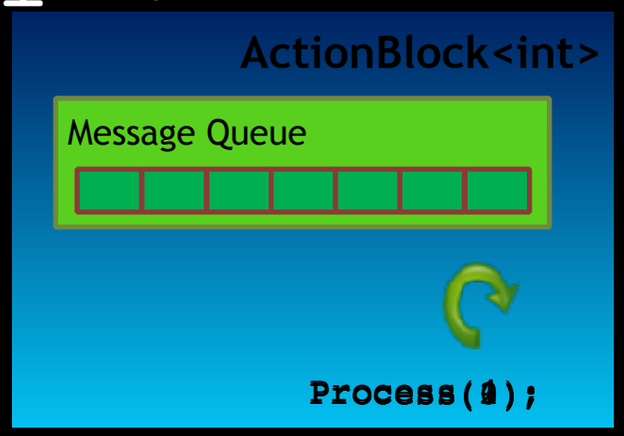



client
client
client
mainReactor
acceptor
subReactor
read
read
read
send
send
send
Thread
Pool
decode compute encode
decode compute encode
worker
threads
decode compute encode
queued tasks

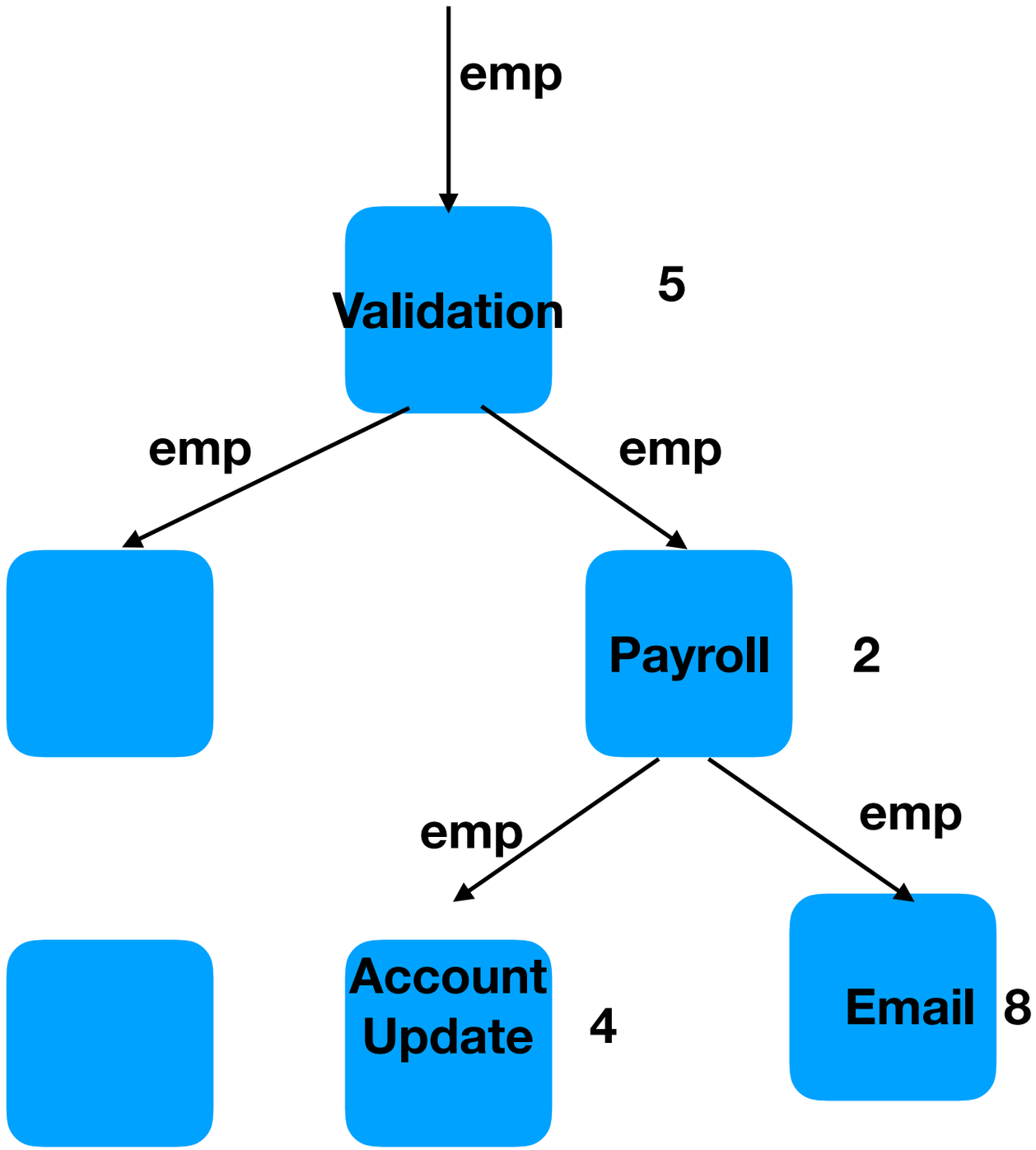


Netty is a low level non-blocking I/O library, used in most reactive solutions including Akka, Vert.x, HBaseAsync, PgAsync.

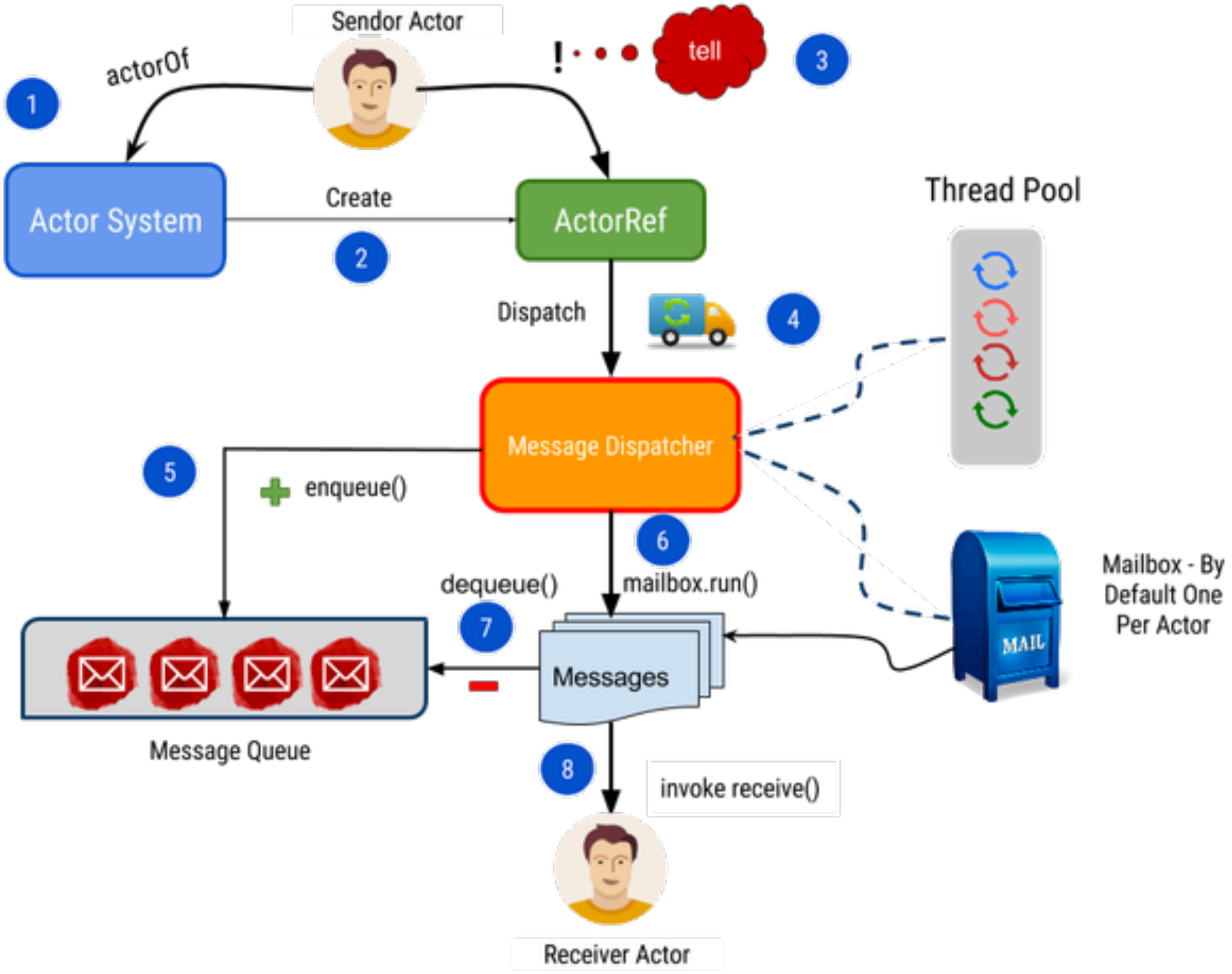
```
var c = new ActionBlock<int>(i =>
{
    Process(i);
});
```




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



async fun()
go fun()



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

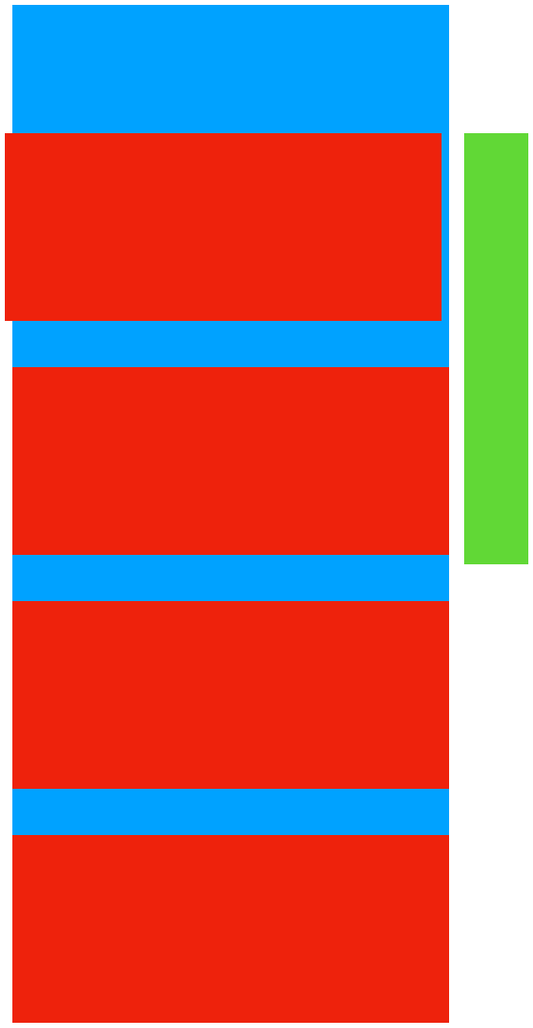
Reactor and Proactor are two event handling patterns

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

Thread-safe collections

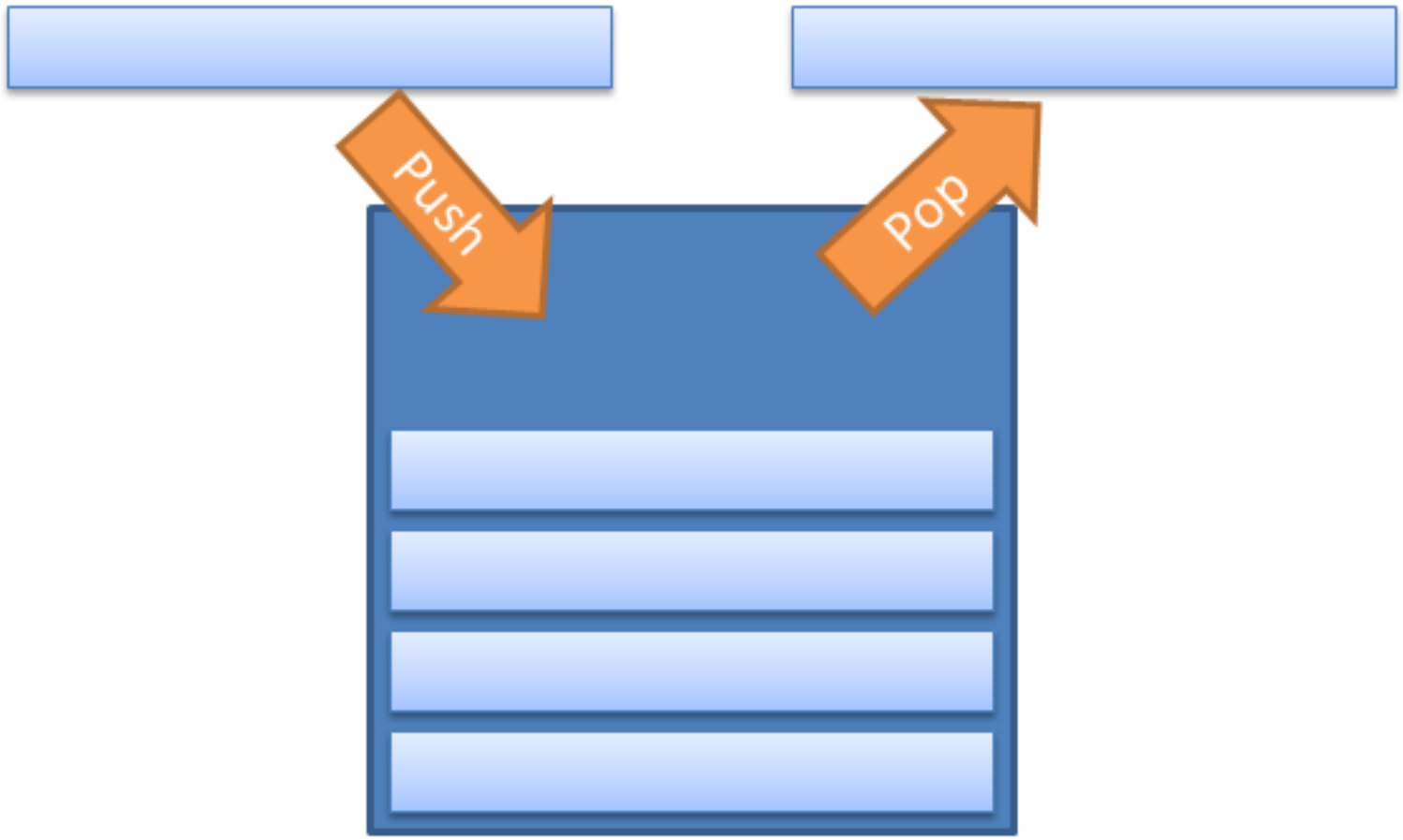
ConcurrentBag<T>

ConcurrentStack<T>

ConcurrentQueue<T>

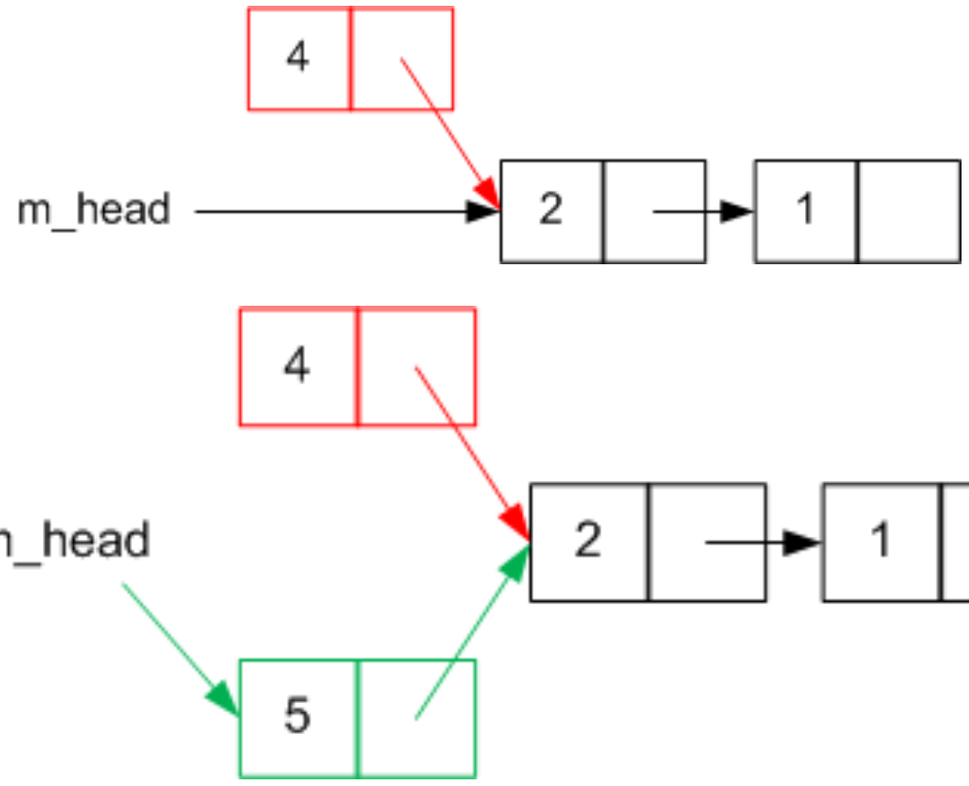
ConcurrentDictionary<TKey,Tvalue>

Concurrent Stack



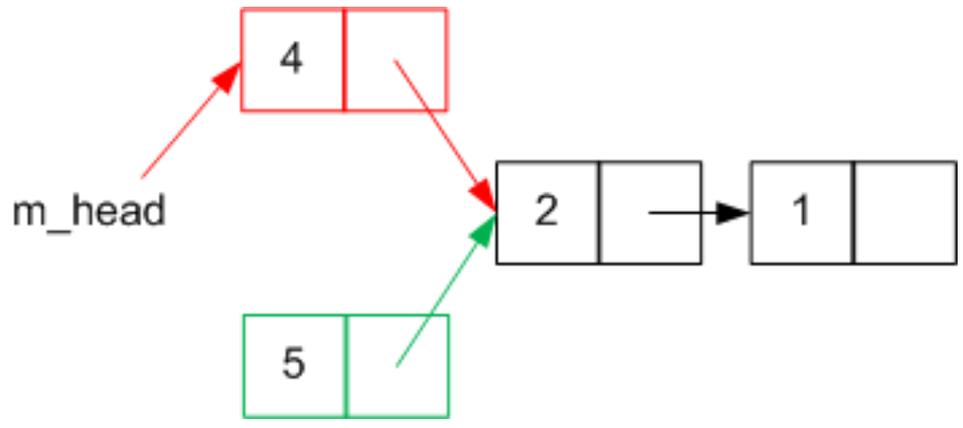
Adding a item to stack

Thread 1, cycle 1 ,
executes 1,2

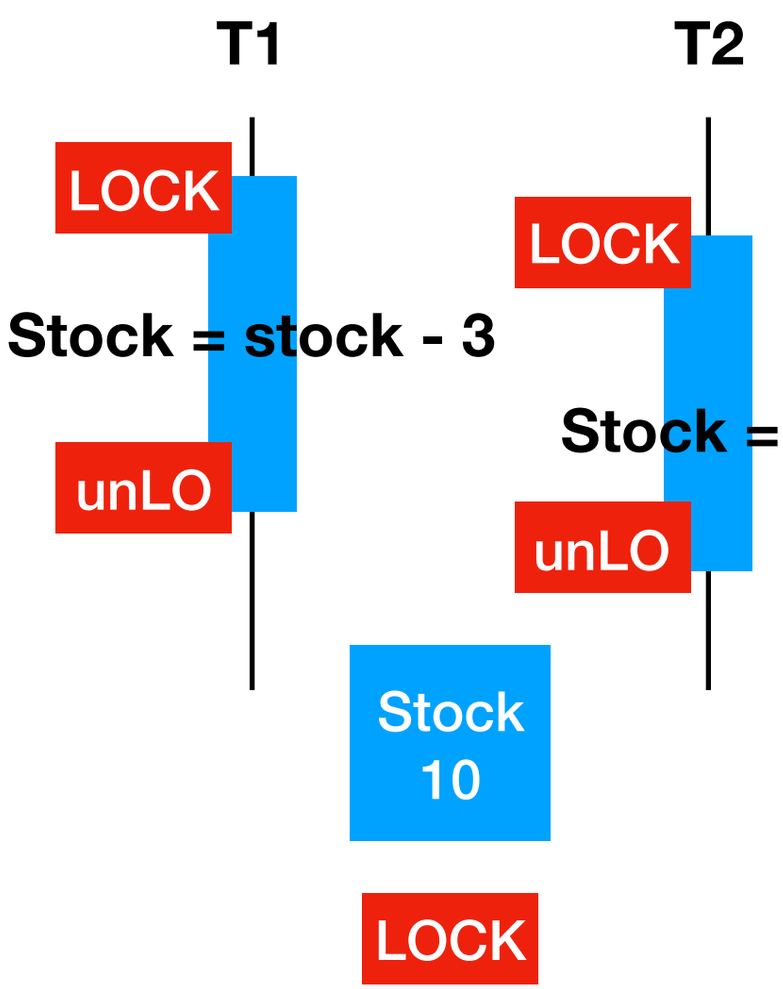


Thread 2, cycle 2 ,
executes 1,2,3

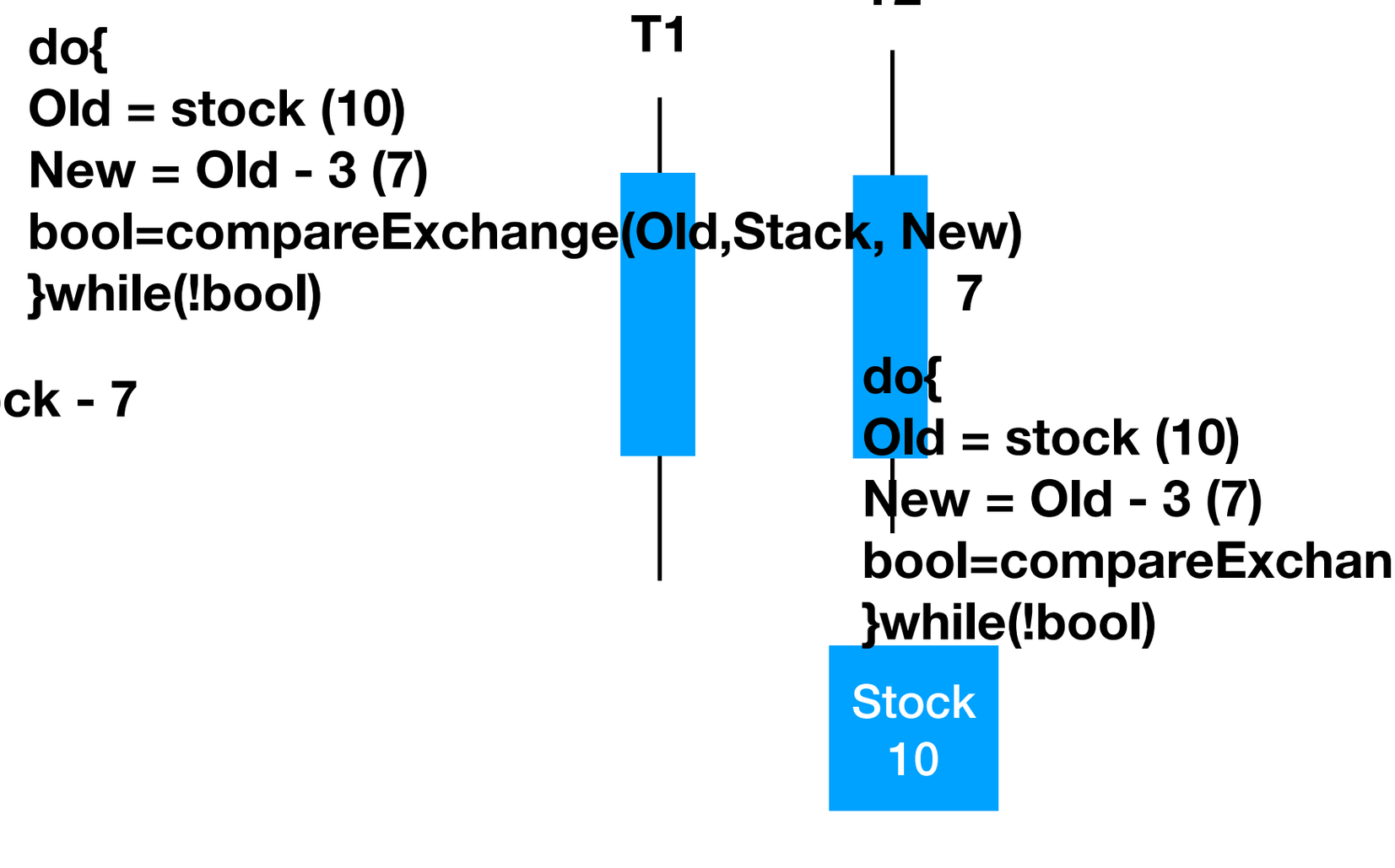
Thread 1, cycle 3 ,
executes 3



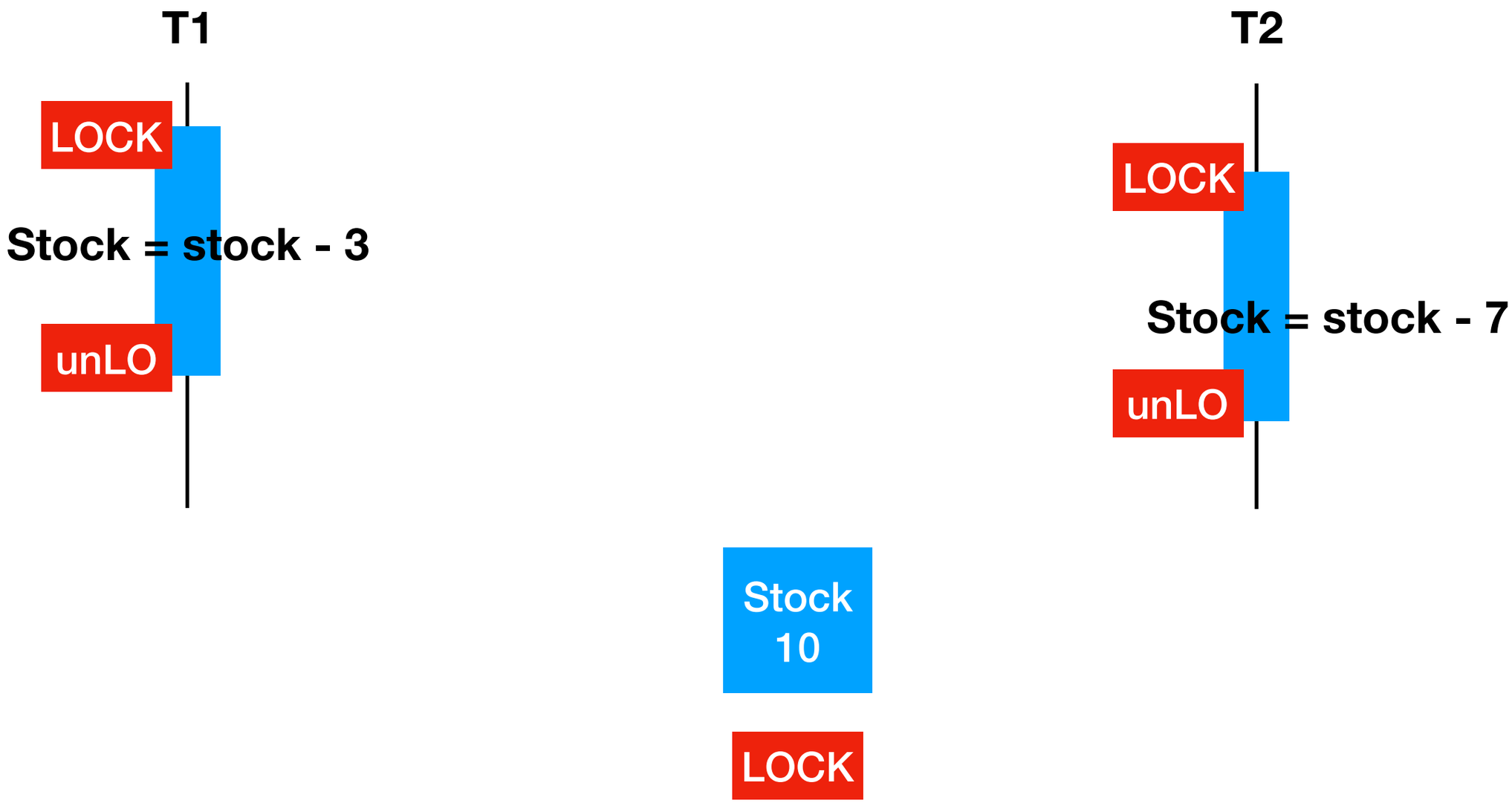
Pessimistic Approach



Optimistic Approach



Pessimistic Approach



Optimistic Approach

T1

```
do{
Old = stock (10)
New = Old - 3 (7)
bool=compareExchange(Old,Stack, New)
if(!bool)
    Spin(10000);
}while(!bool)
```

3

T2

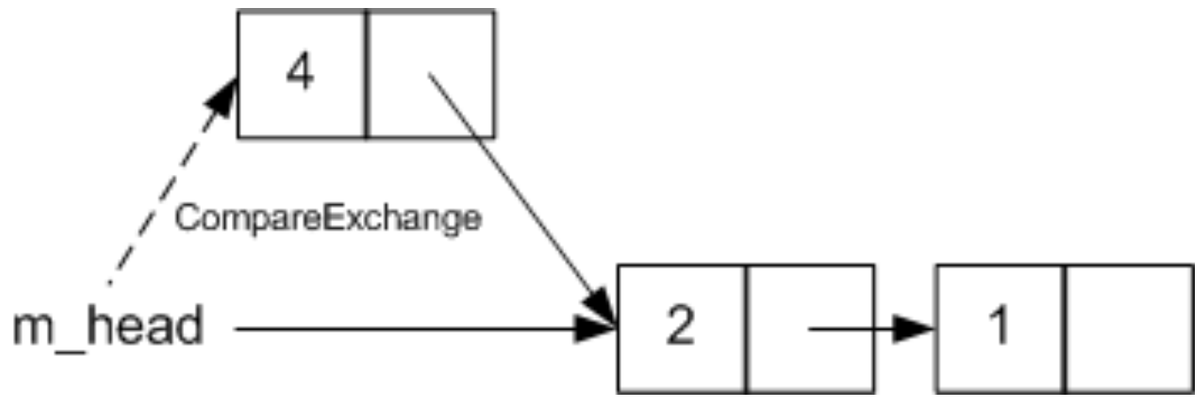
```
do{
Old = stock (10)
New = Old - 7 (3)
bool=compareExchange(Old,Stack, New)
}while(!bool)
```

7

Stock
10

Adding a item to Concurrent Stack

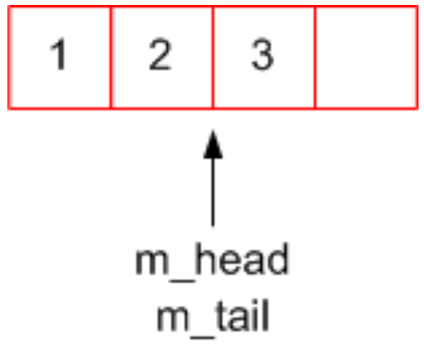
```
public void Push(T item)
{
    Node node = new Node(item);       //step 1
    do {
            node.m_next = m_head;     //step 2
    } while (
    Interlocked.CompareExchange(ref m_head, node, node.m_next) !=
    node.m_next);                                   //step 3
}
```



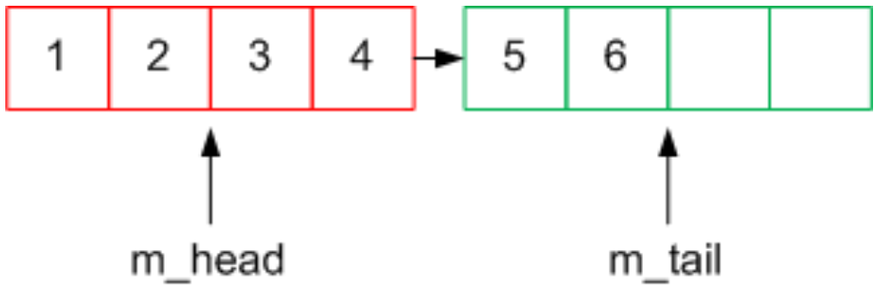
Perform step 3 *only if the head hasn't changed in the meantime*. If it has been changed, then go back to step 2 to use the updated head node as the value of m_next.

ConcurrentQueue

ConcurrentQueue

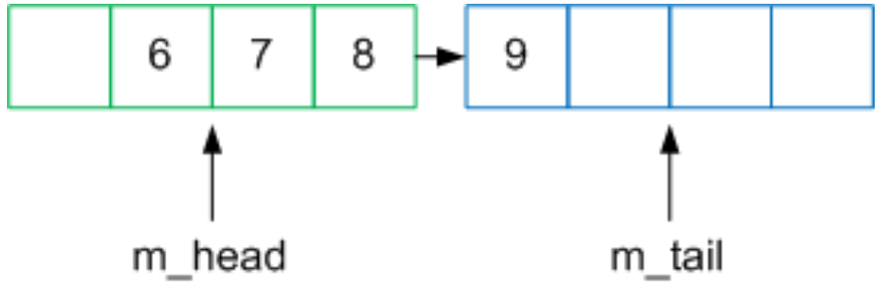


three items are enqueued



Three more items are then enqueued

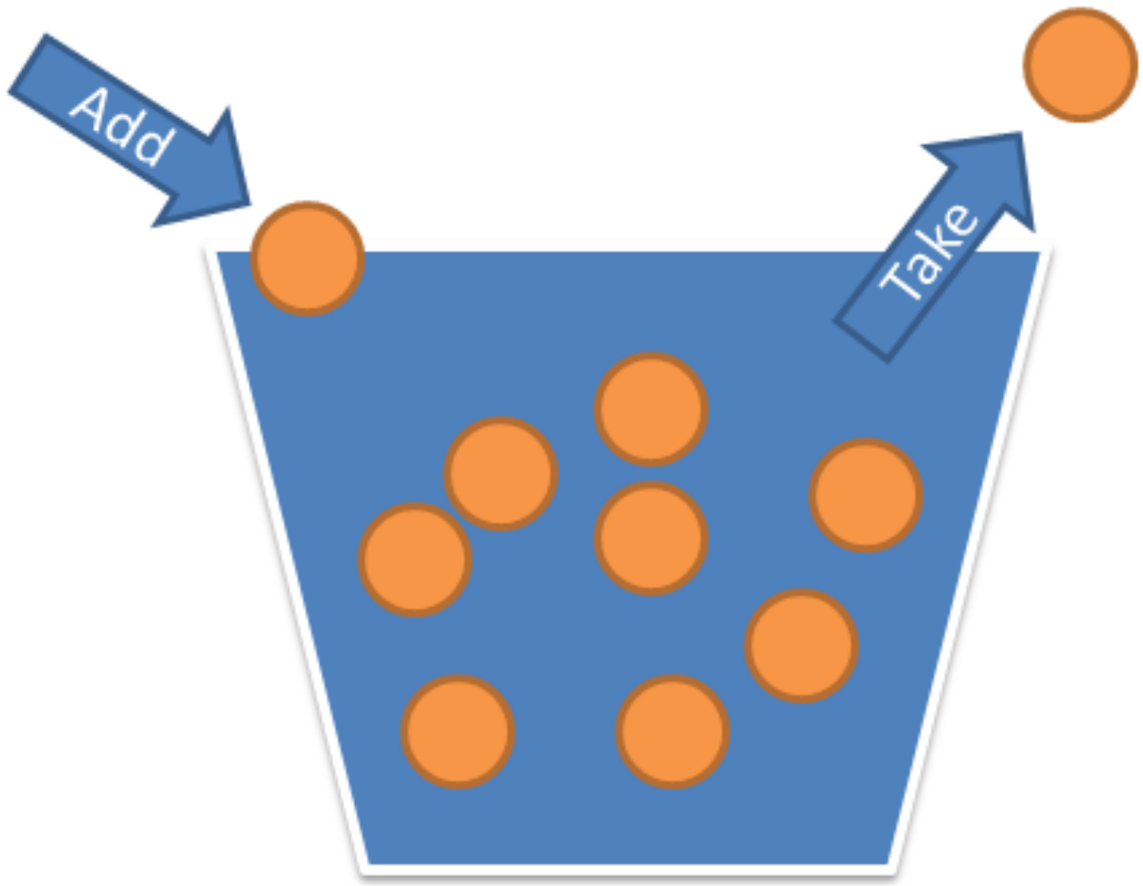
And another three



Five items are then dequeued from the head.

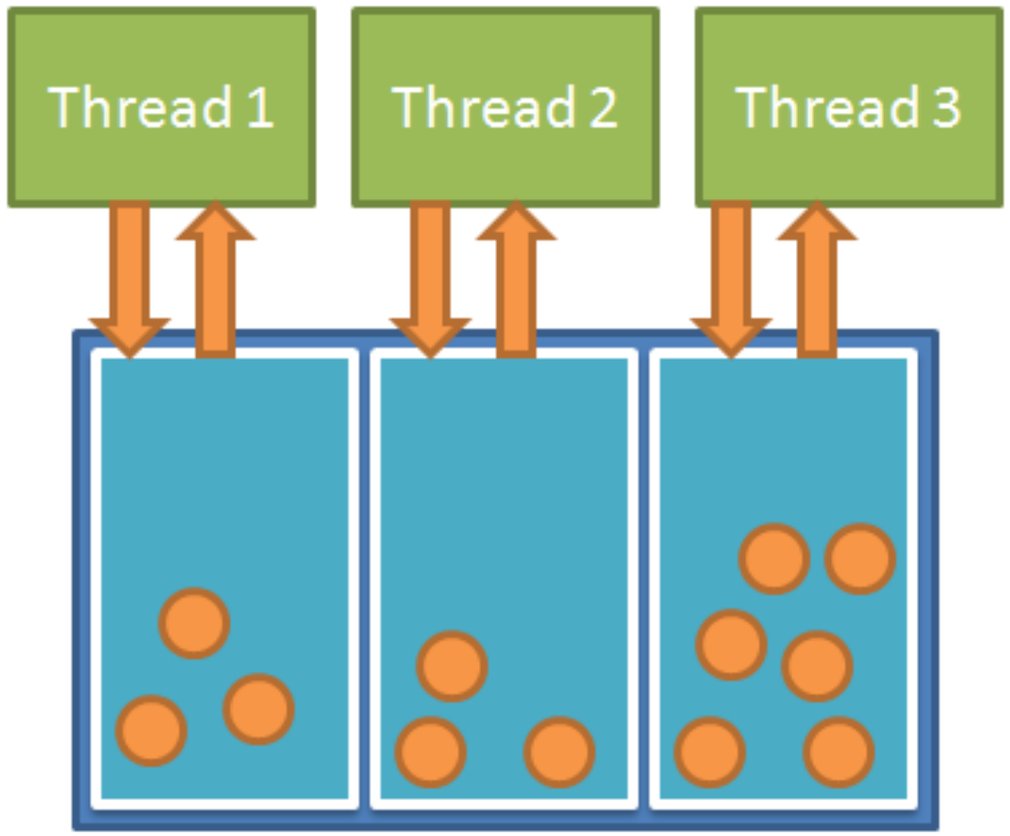
ConcurrentBag

ConcurrentHashMultiset



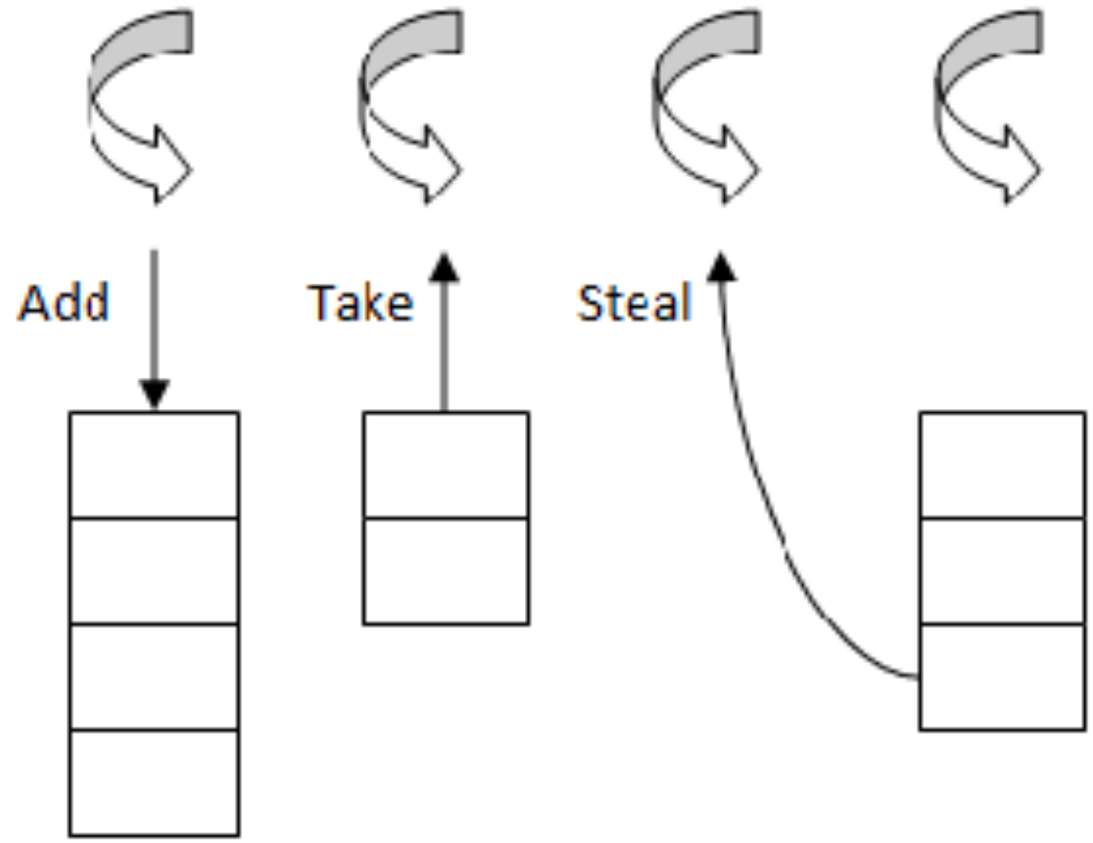
a thread-safe, unordered collection of objects

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

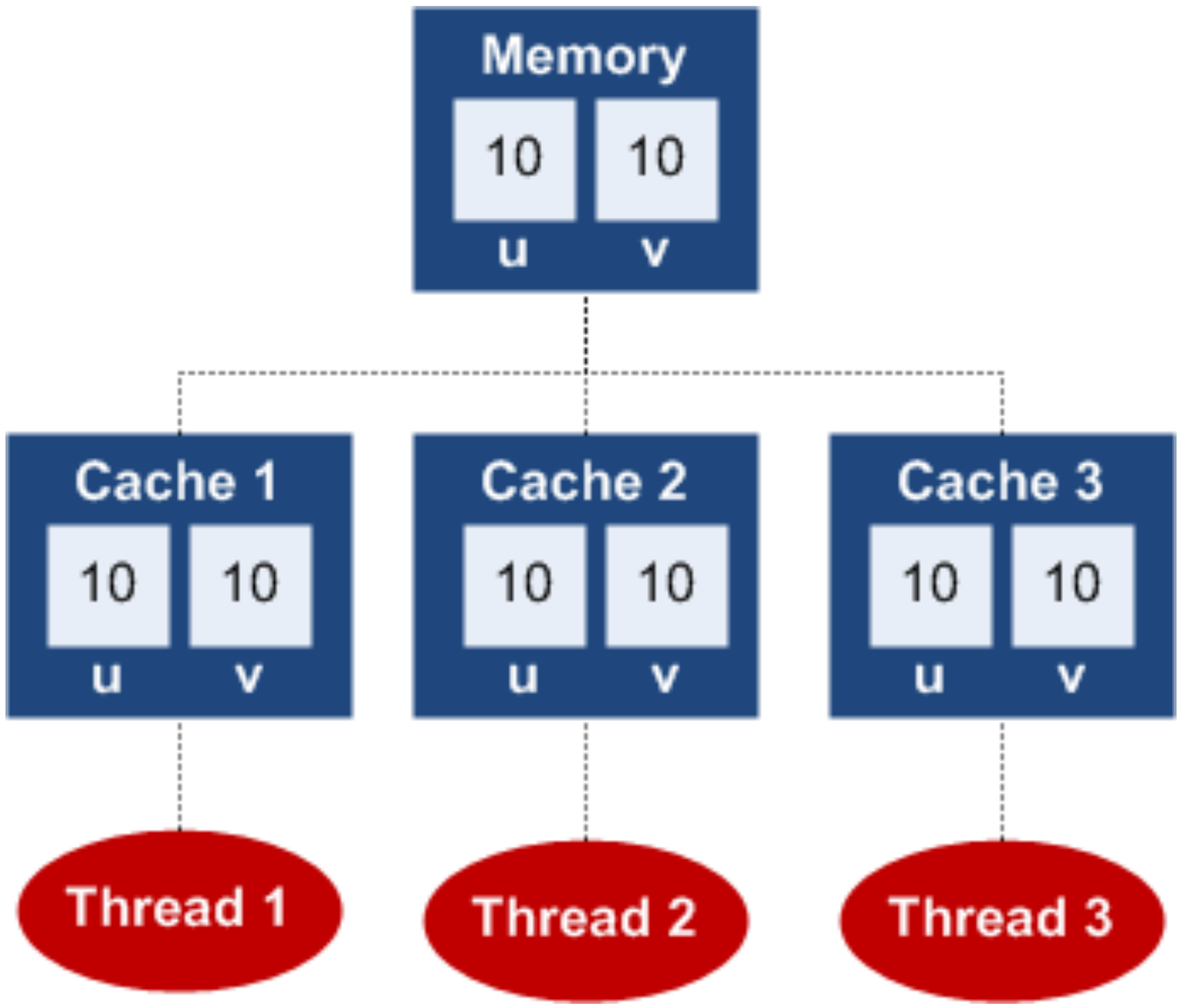
```
class Foo
{
    int _answer=100;
    bool _complete=false;
    void A()
    {
        _answer = 123;
        _complete = true;
    }
    void B()
    {
        if(_complete)
        {
            Console.WriteLine (_ answer);
        }
    }
}
```

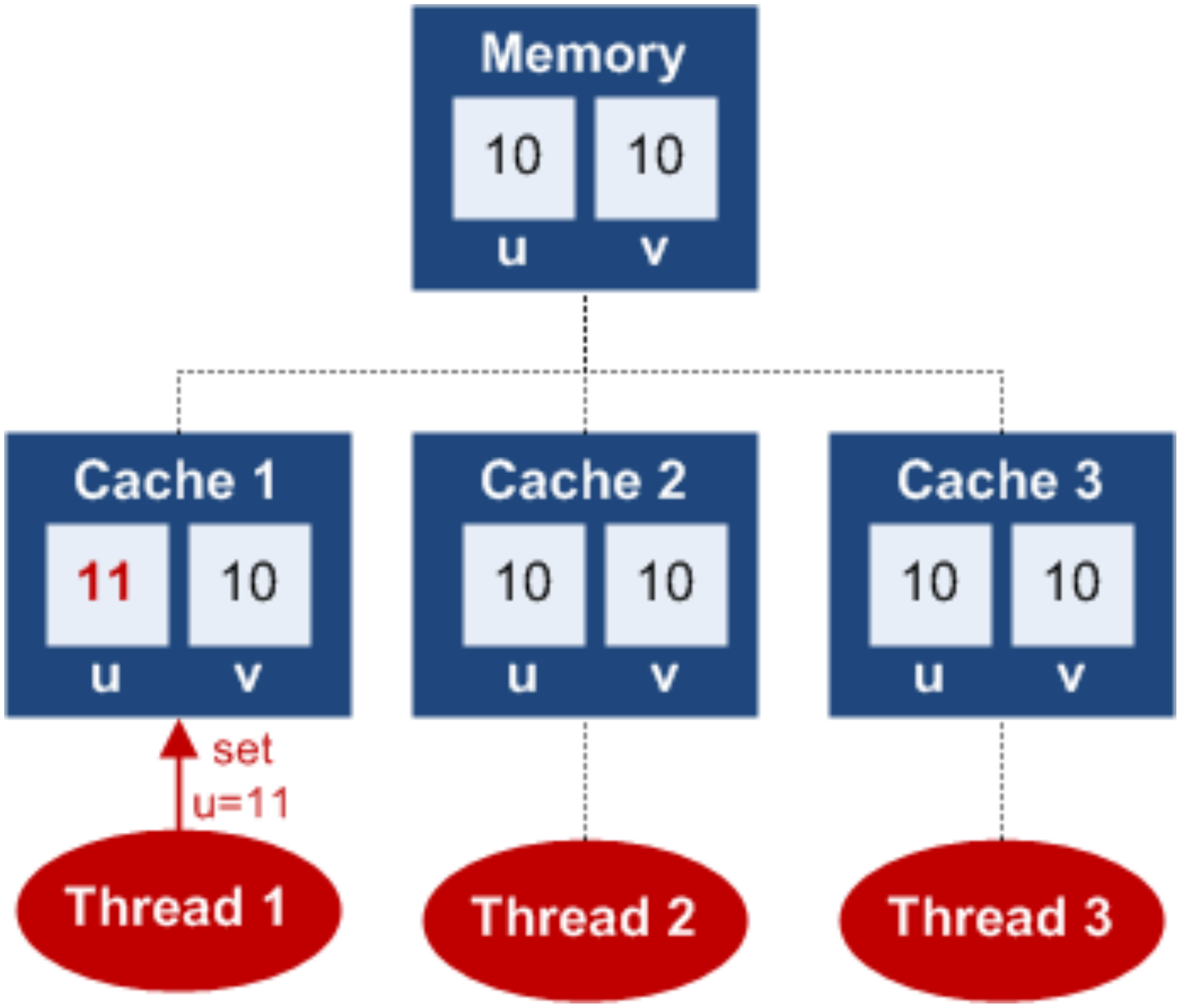
```
class Foo
{
    int _answer;
    bool _complete;
    void A()
    {
            _answer = 123;
            _complete = true; //←   instructions could get reordered causing B to write
    "0"
    }
    void B()
    {
            if (_complete)
                    Console.WriteLine (_answer);
    }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

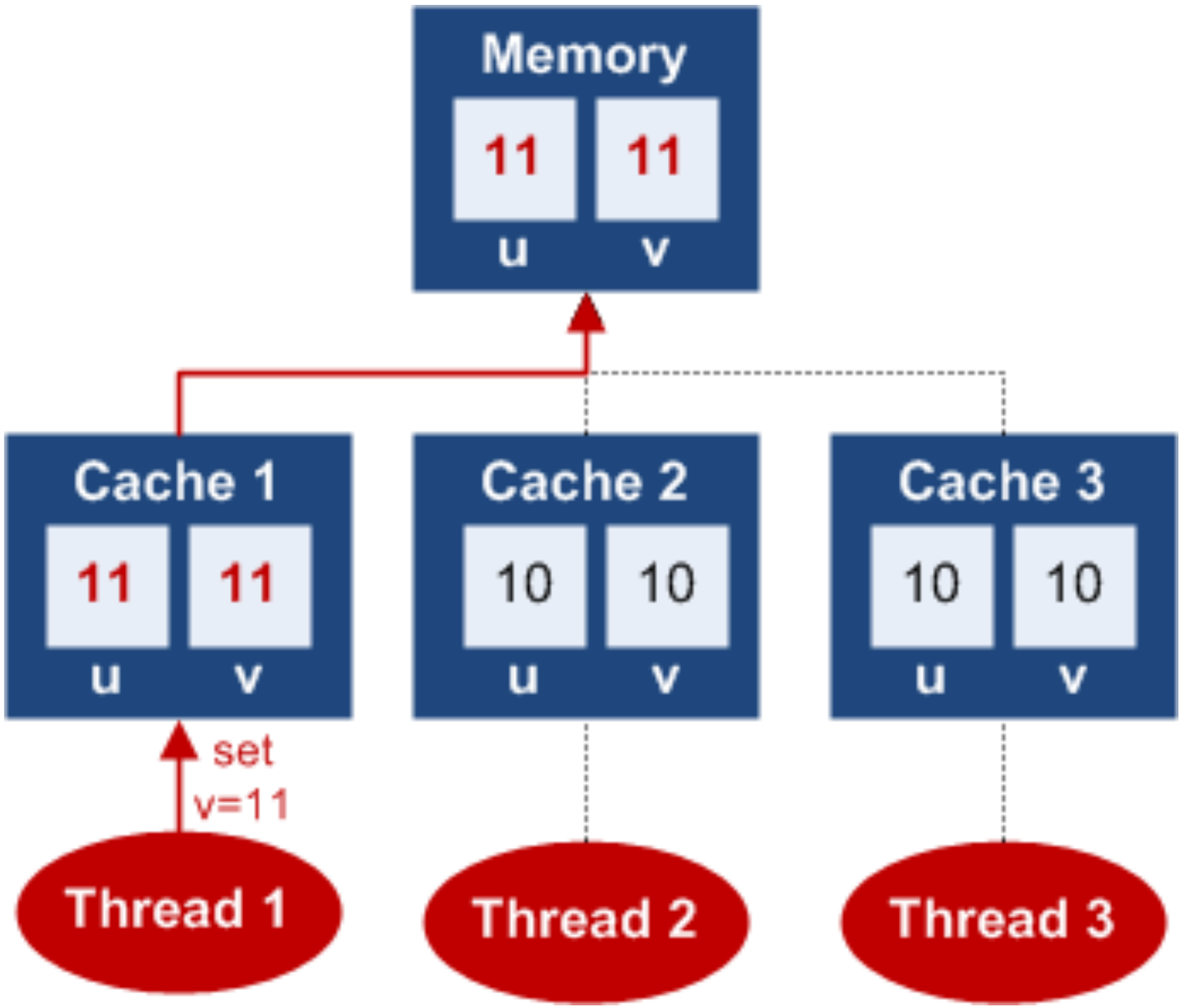
```
static void Main()
{
        bool complete = false;
         var t = new Thread(() =>
         {
            int i=0;
            while (!complete)
           {
                    Thread.MemoryBarrier();
                    i++;
            }

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

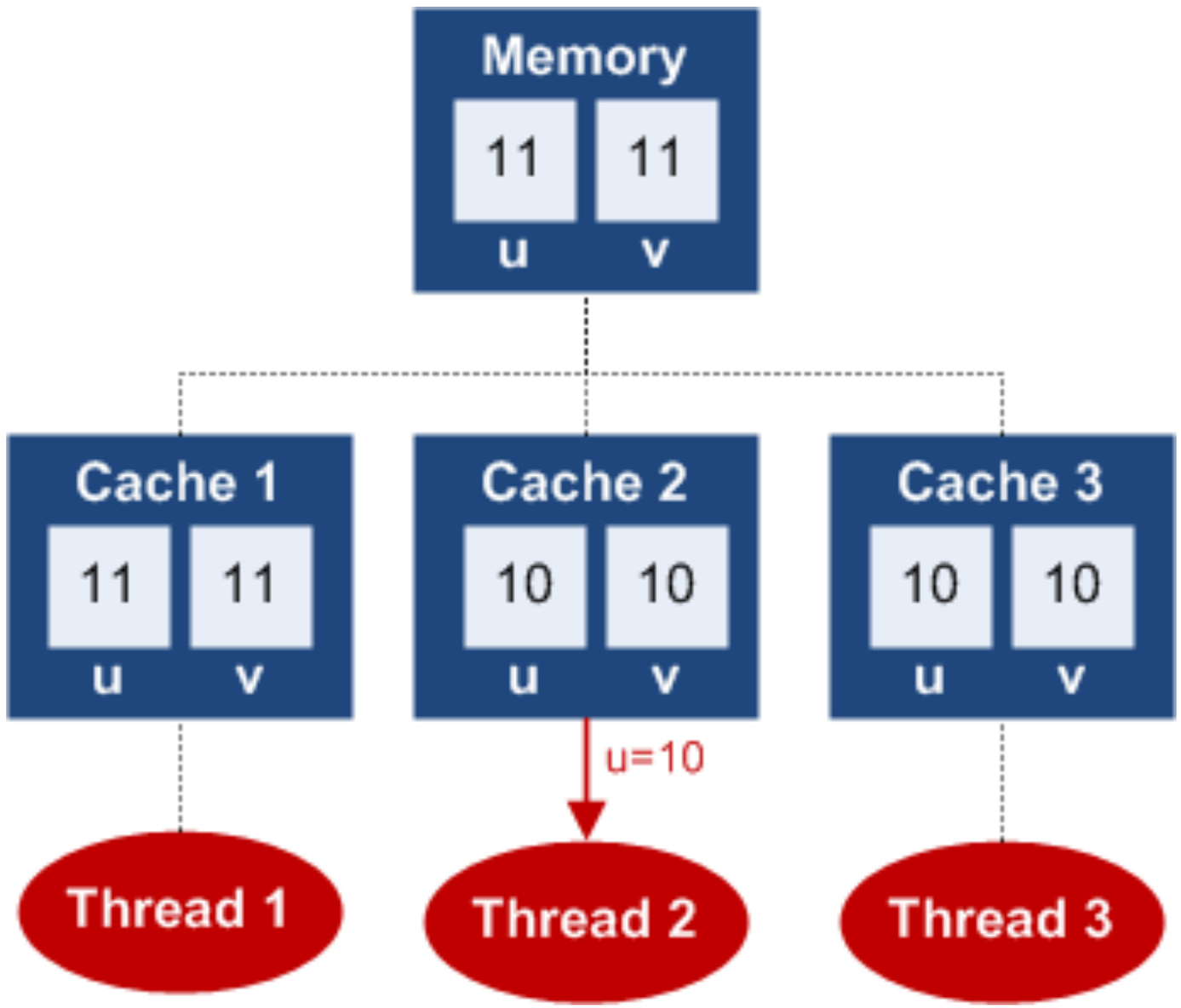
run it with optimizations enabled and Release Mode

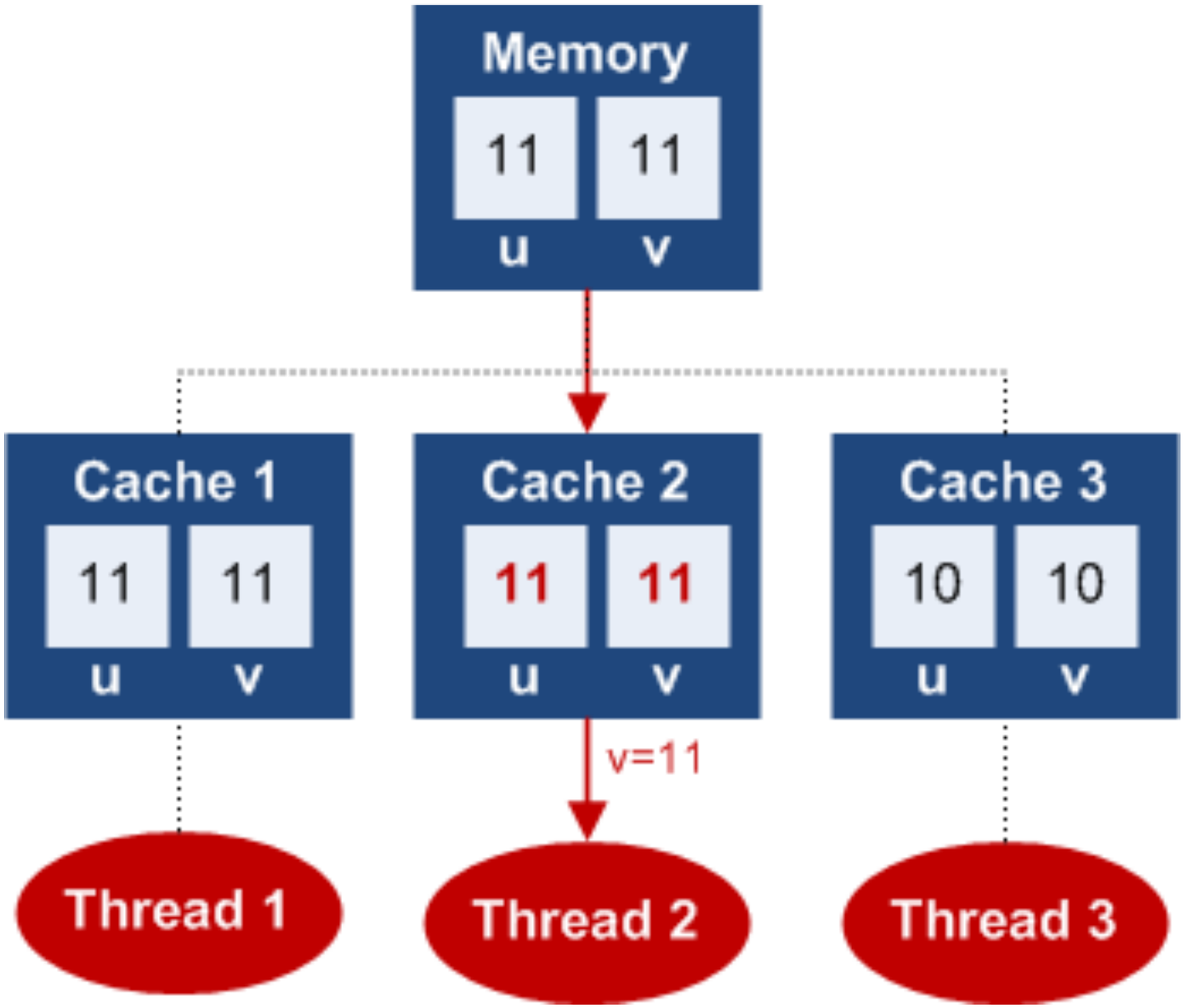

Memory
10
10
u
v
Cache 1
10
10
u
v
Cache 2
10
10
u
v
Cache 3
10
10
u
v
Thread 1
Thread 2
Thread 3


Memory
10
10
u
v
Cache 1
11
10
u
v
Cache 2
10
10
u
v
Cache 3
10
10
u
v
set
u=11
Thread 1
Thread 2
Thread 3



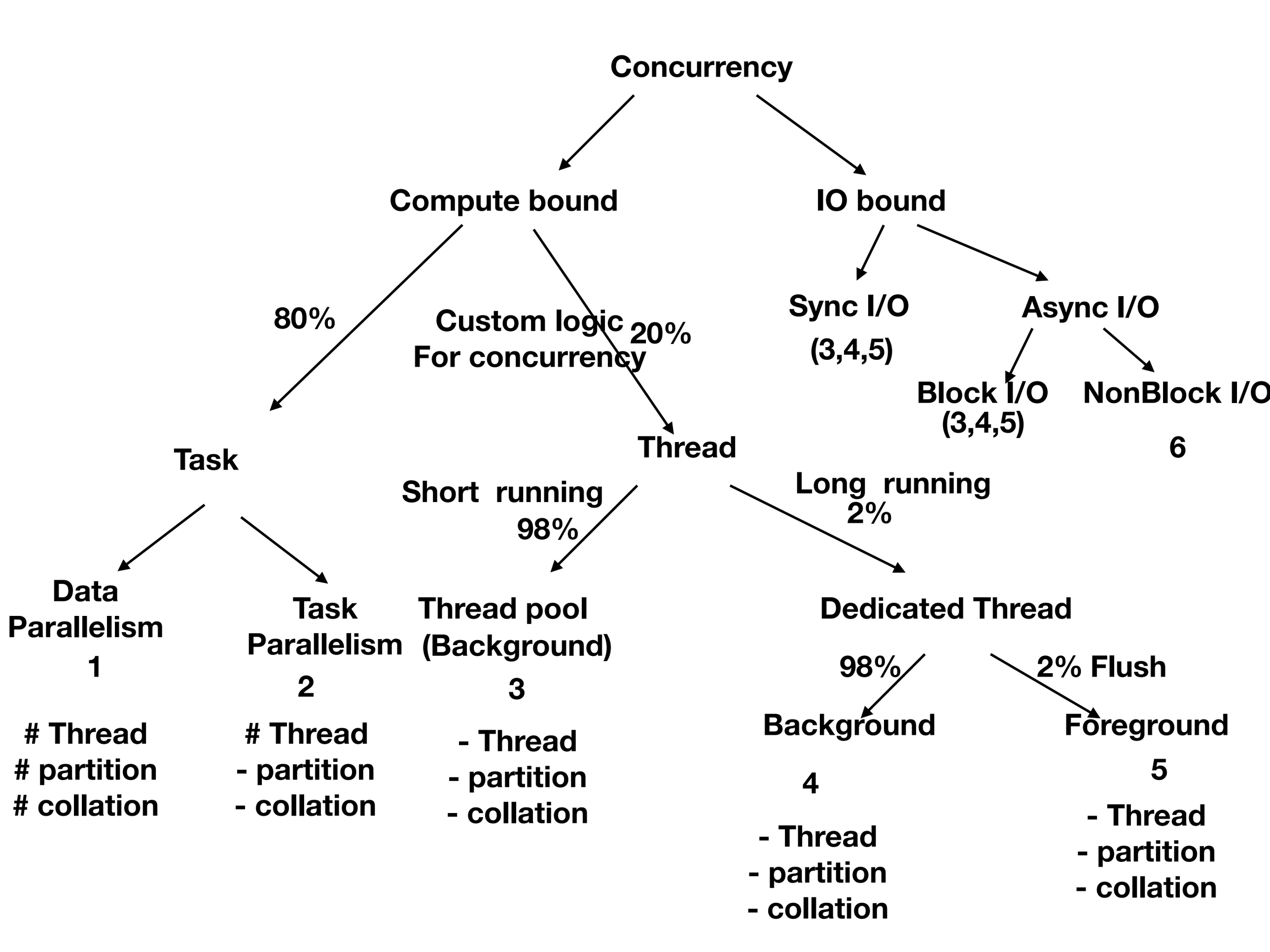
all C# writes are volatile


Memory
11
11
u
v
Cache 1
11
11
u
v
Cache 2
10
10
u
v
Cache 3
10
10
u
v
u=10
Thread 1
Thread 2
Thread 3


Memory
11
11
u
v
Cache 1
11
11
u
v
Cache 2
11
11
u
v
v=11
Cache 3
10
10
u
v
Thread 1
Thread 2
Thread 3

Synchronization constructs

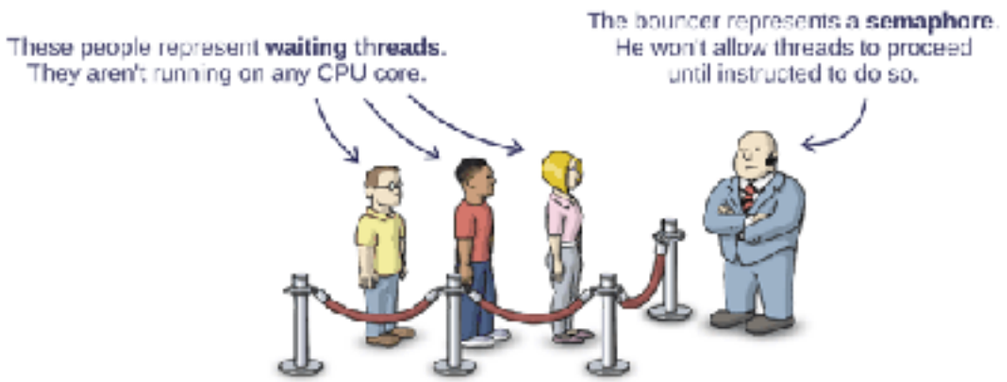
- **Simple blocking methods**
  - These wait for another thread to finish or for a period of time to elapse.
- **Locking constructs**
  - These limit the number of threads that can perform some activity or execute a section of code at a time.
- **Signaling constructs**
  - These allow a thread to pause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

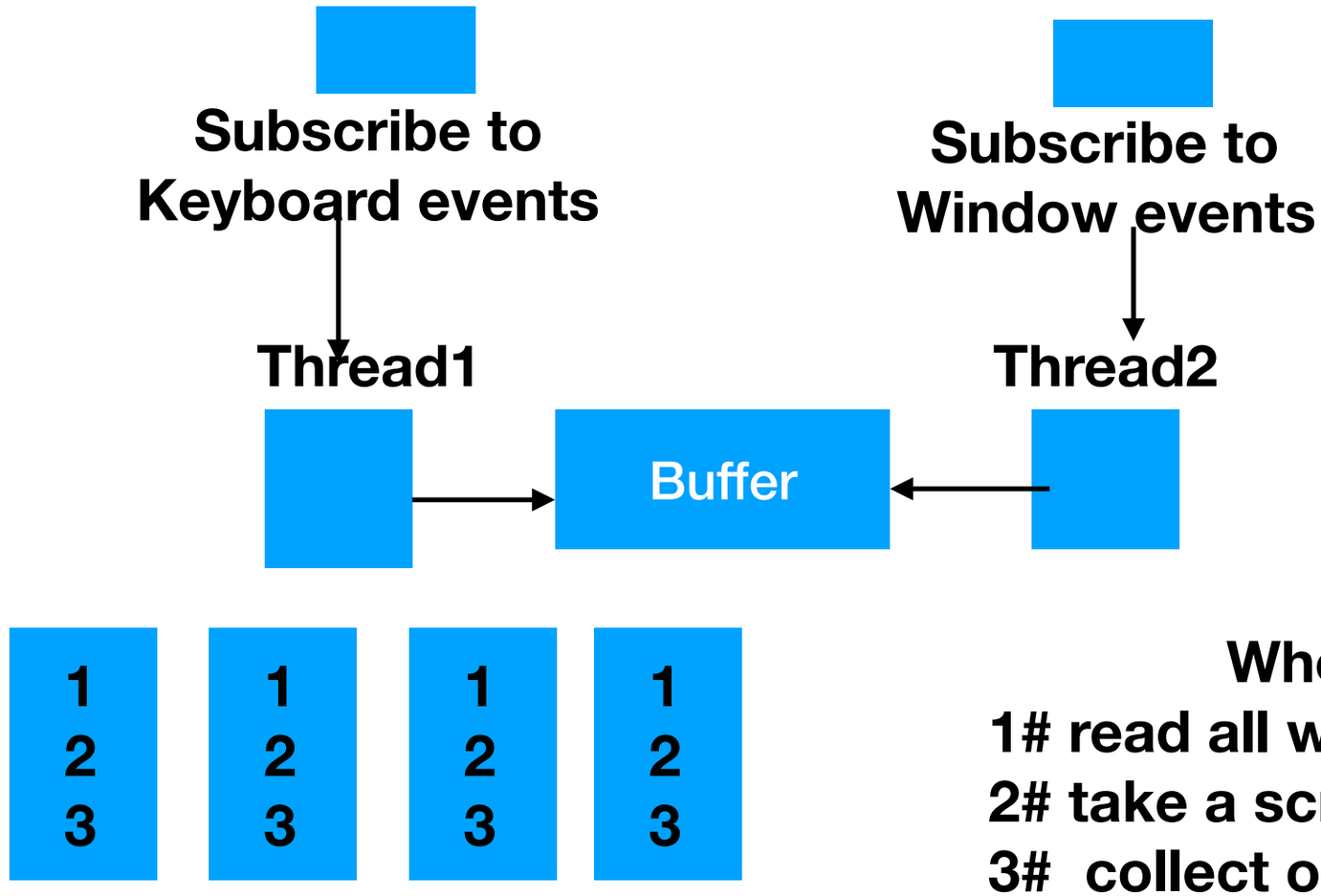

Concurrency
Compute bound
IO bound
80%
Custom logic
For concurrency
20%
Sync I/O
(3,4,5)
Async I/O
Block I/O
(3,4,5)
NonBlock I/O
6
Task
Thread
Short running
98%
Long running
2%
Data
Parallelism
1
Thread
partition
collation
Task
Parallelism
2
Thread
- partition
- collation
Thread pool
(Background)
3
- Thread
- partition
- collation
Dedicated Thread
98%
2% Flush
Background
4
- Thread
- partition
- collation
Foreground
5
- Thread
- partition
- collation

Syncronization

- **Nonblocking synchronization constructs**
  - volatile
  - Atomic.
- **Simple blocking methods**
  - Sleep (bad)
  - Join
  - Spin
  - Spin(10000)
- **Locking constructs**
  - *Exclusive* locking
  - nonexclusive locking
    - Semaphore
    - reader/writer locks.
- **Signaling constructs**
  - event wait
  - Latch
  - Barrier

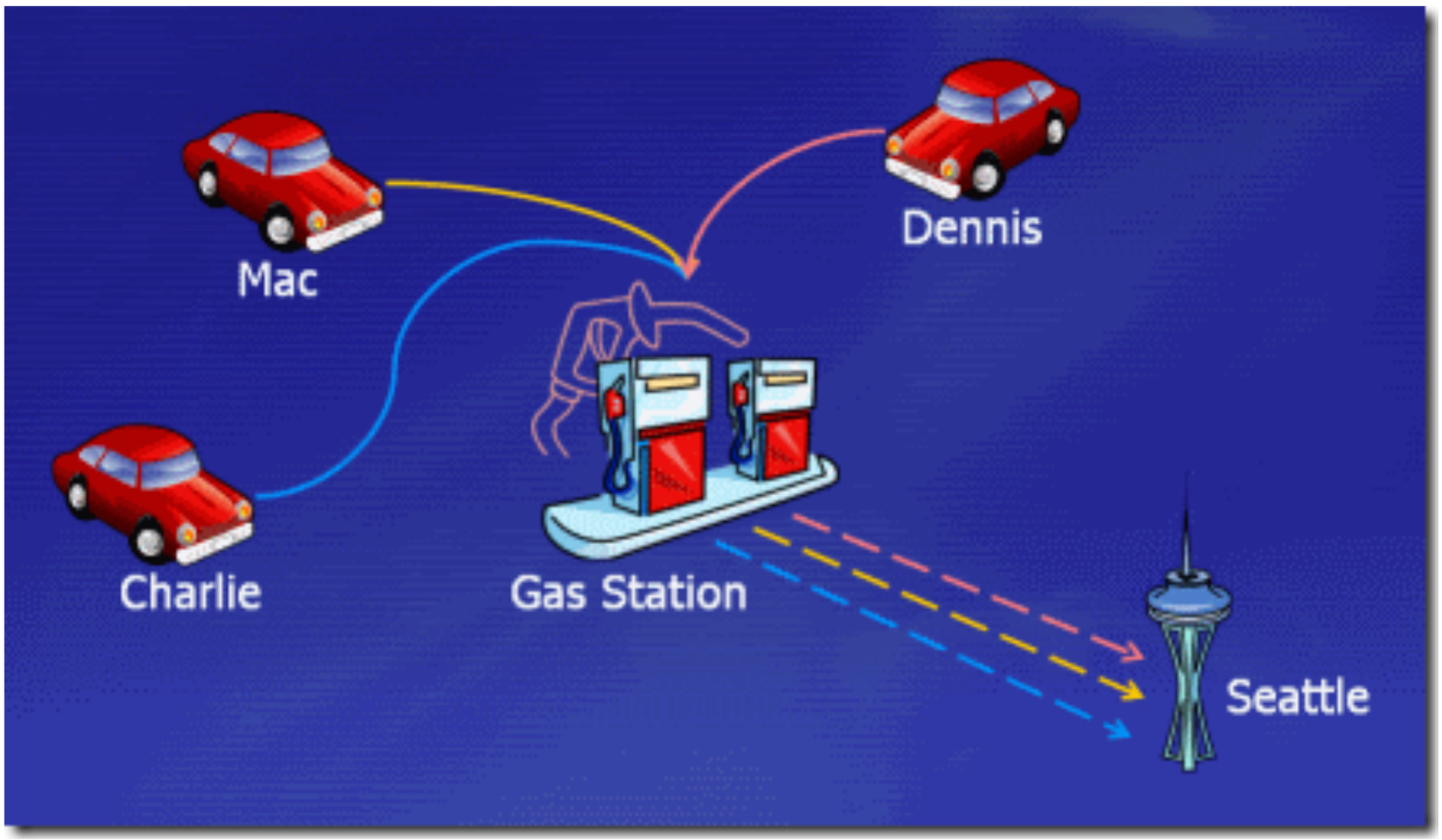
- **Nonblocking synchronization constructs**
  - volatile
  - Atomic.
- **Simple blocking methods (For a Period)**
  - Sleep (bad)
  - Join
  - Spin
- **Locking constructs**
  - *Exclusive* locking
  - nonexclusive locking
    - Semaphore
    - reader/writer locks.
- **Signaling constructs**
  - event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





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

Latch vs Barrier



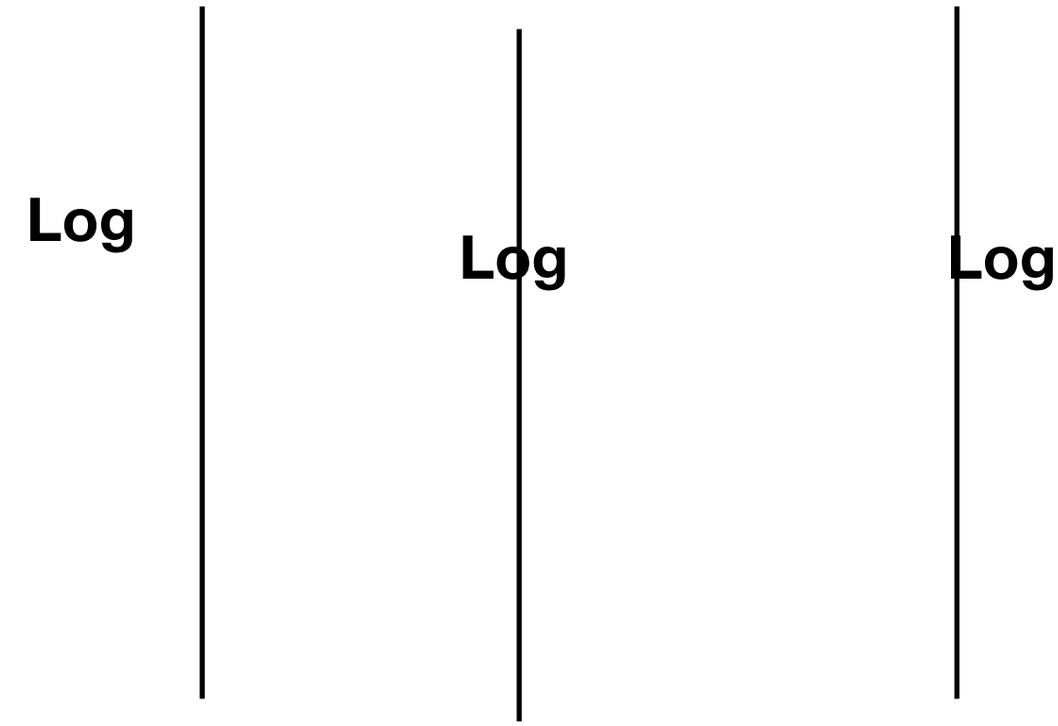
Latches are for waiting for events; barriers are for waiting for other threads.

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

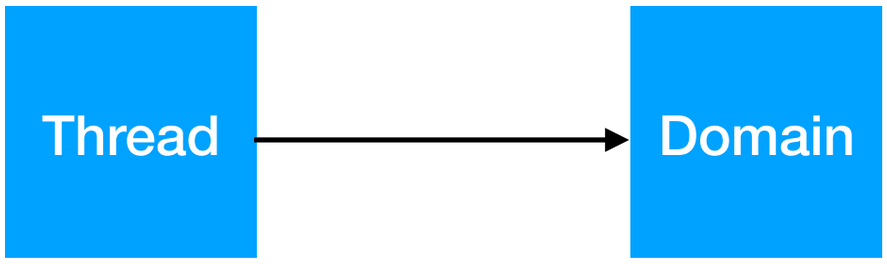
CyclicBarrier, no. of threads that should call await()

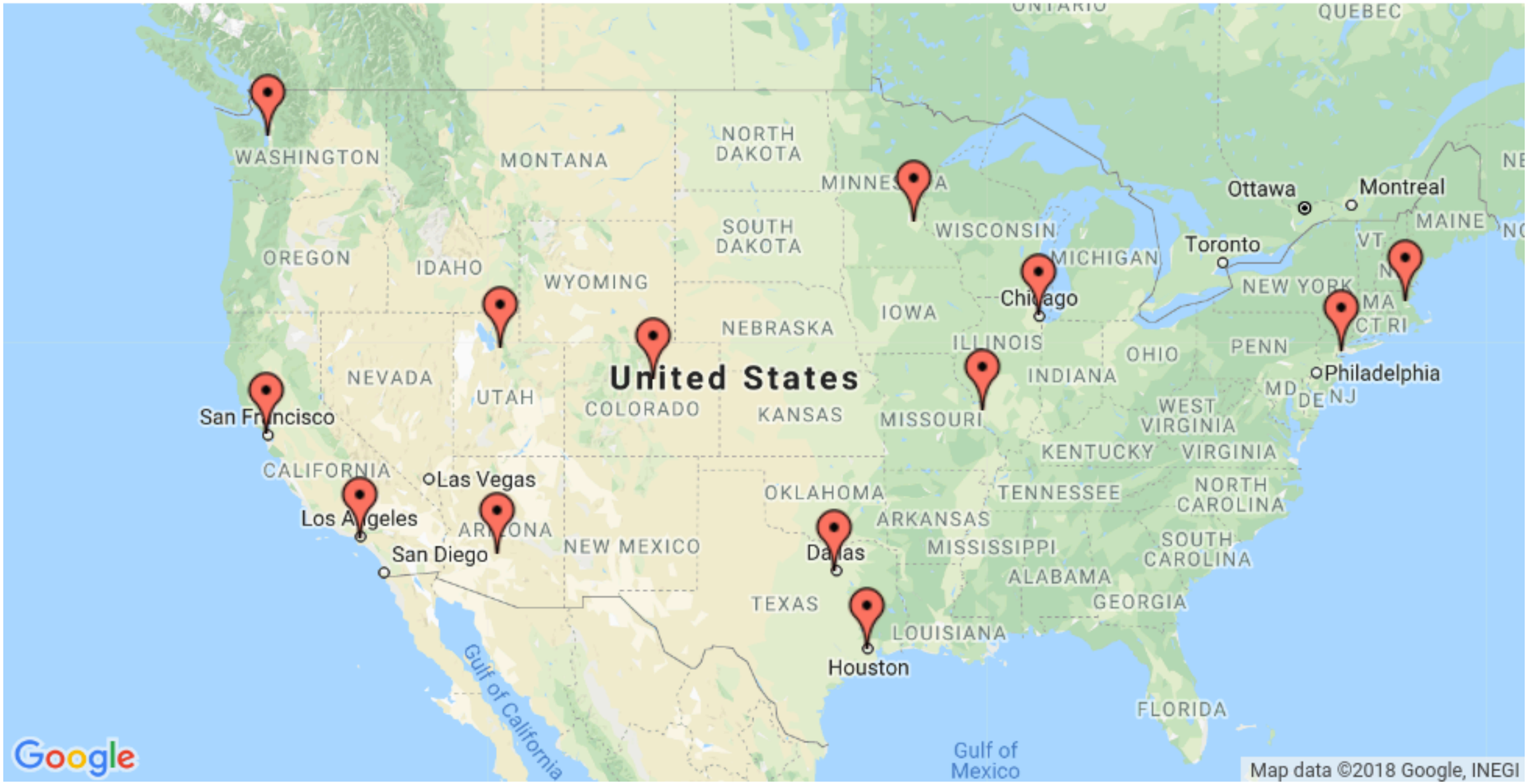
The deprecated Suspend and Resume methods have two modes: dangerous and useless!

CPU profiler
Memory profiler
Thread Profiler



Humble Objects





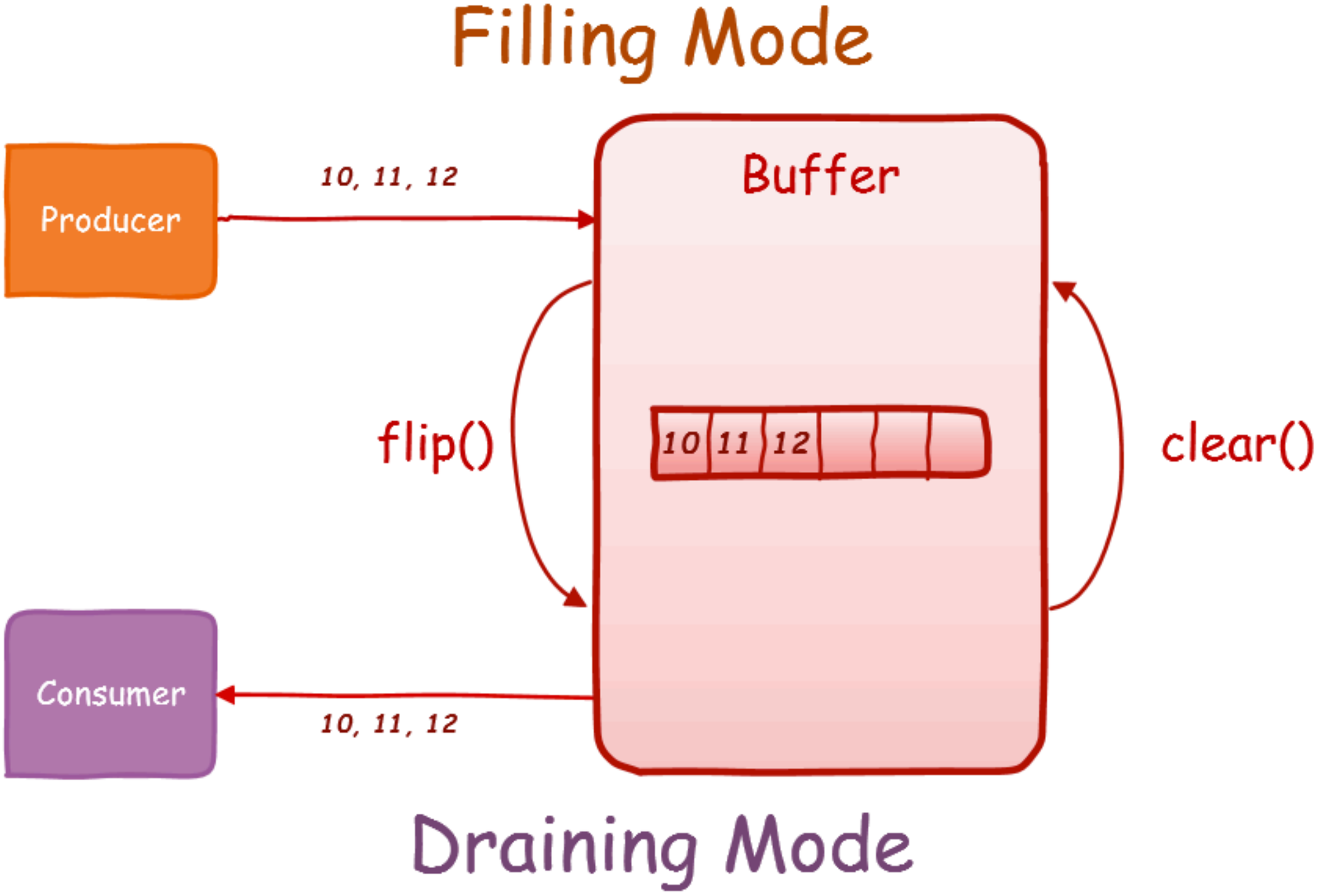
0. New York - 1. Los Angeles - 2. Chicago - 3. Minneapolis - 4. Denver - 5. Dallas - 6. Seattle - 7. Boston - 8. San Francisco - 9. St. Louis - 10. Houston - 11. Phoenix - 12. Salt Lake City

```
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.

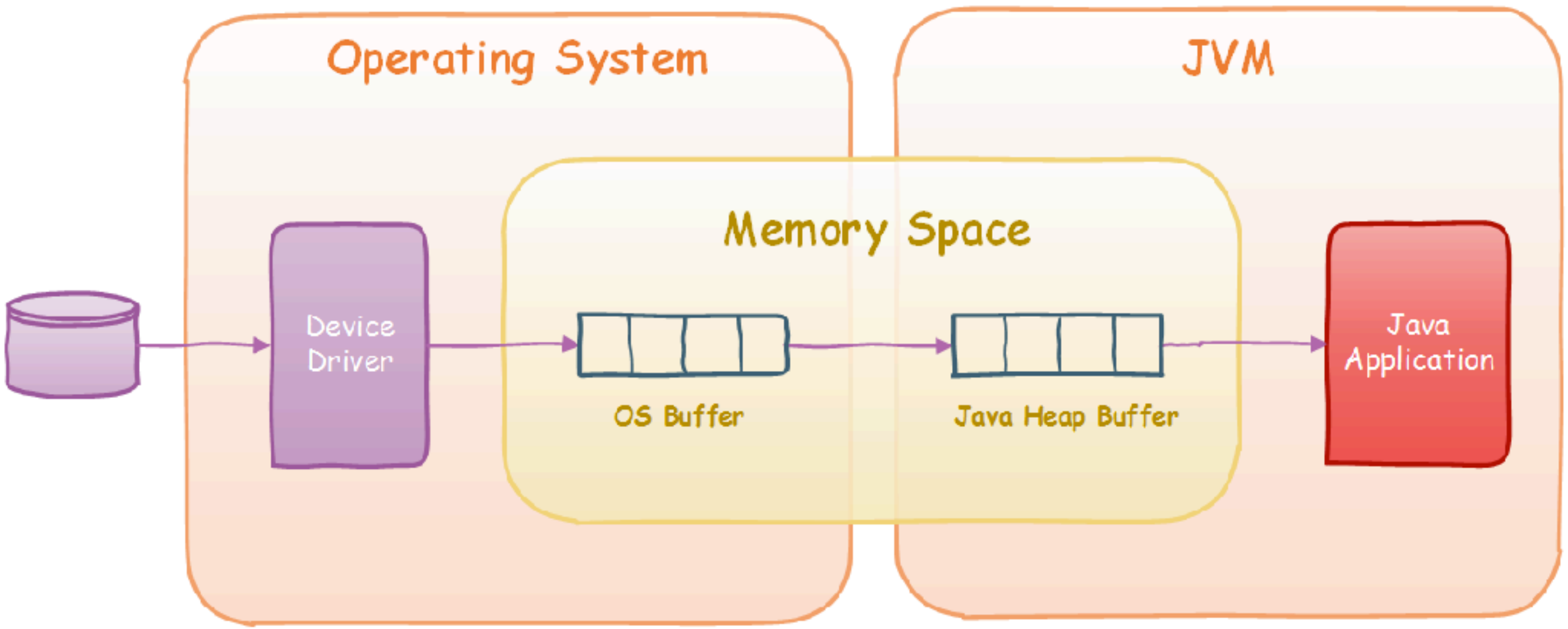
TSP with 10 cities can be solved by a DP method in almost 0.2 seconds using intel core i7. This number increases to almost 13 seconds (~60 times greater) with 15 cities.

Appendix



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);