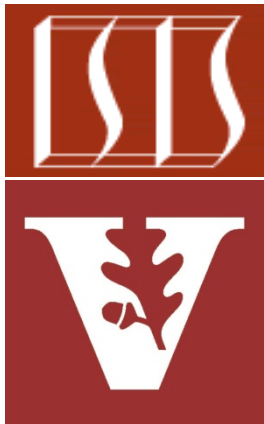


# Android Concurrency: The Monitor Object Pattern (Part 1)



Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

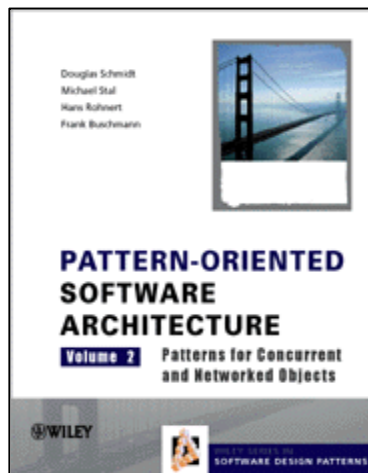
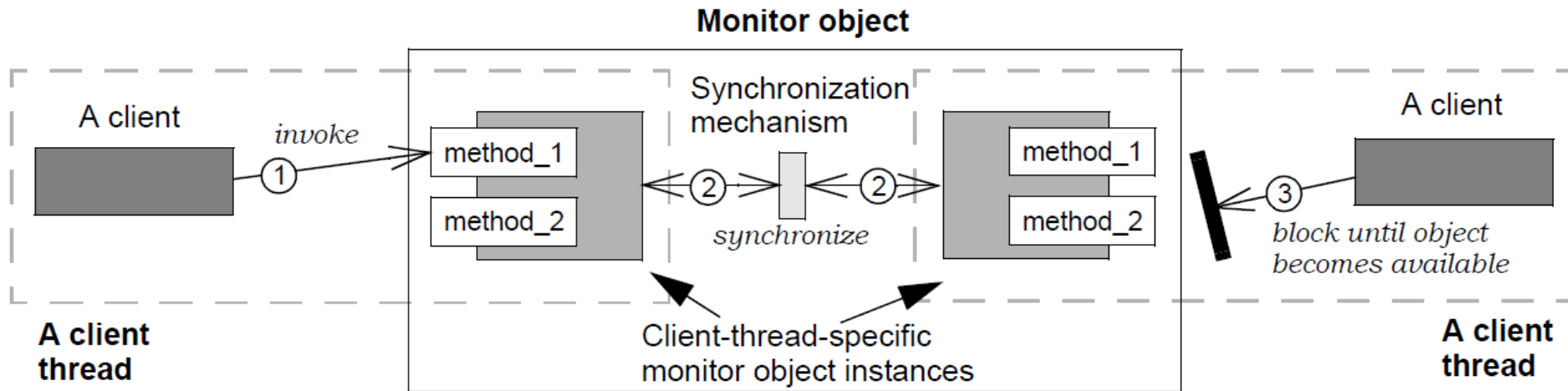
[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Institute for Software  
Integrated Systems  
Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Module

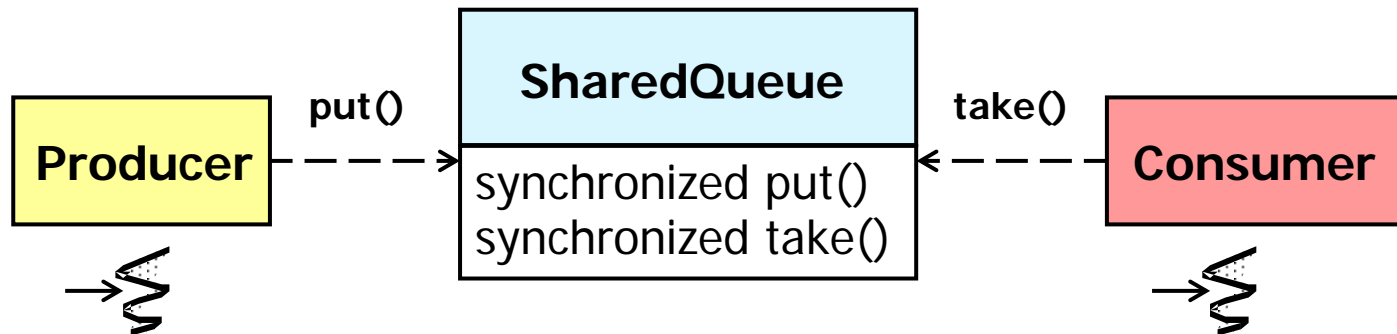
- Understand the *Monitor Object* pattern



# Challenge: Implementing a Blocking Queue

## Context

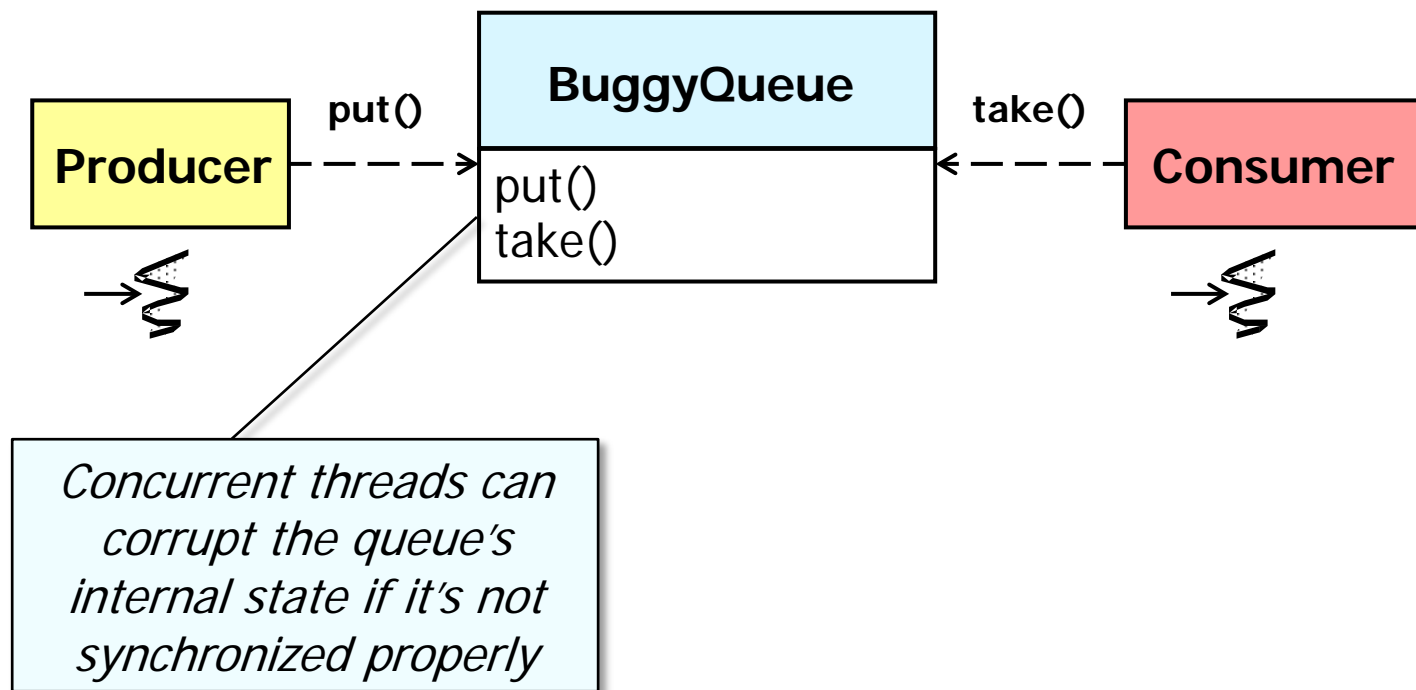
- Concurrent apps/services that need to coordinate interactions between producer & consumer threads via a shared queue



# Challenge: Implementing a Blocking Queue

## Problems

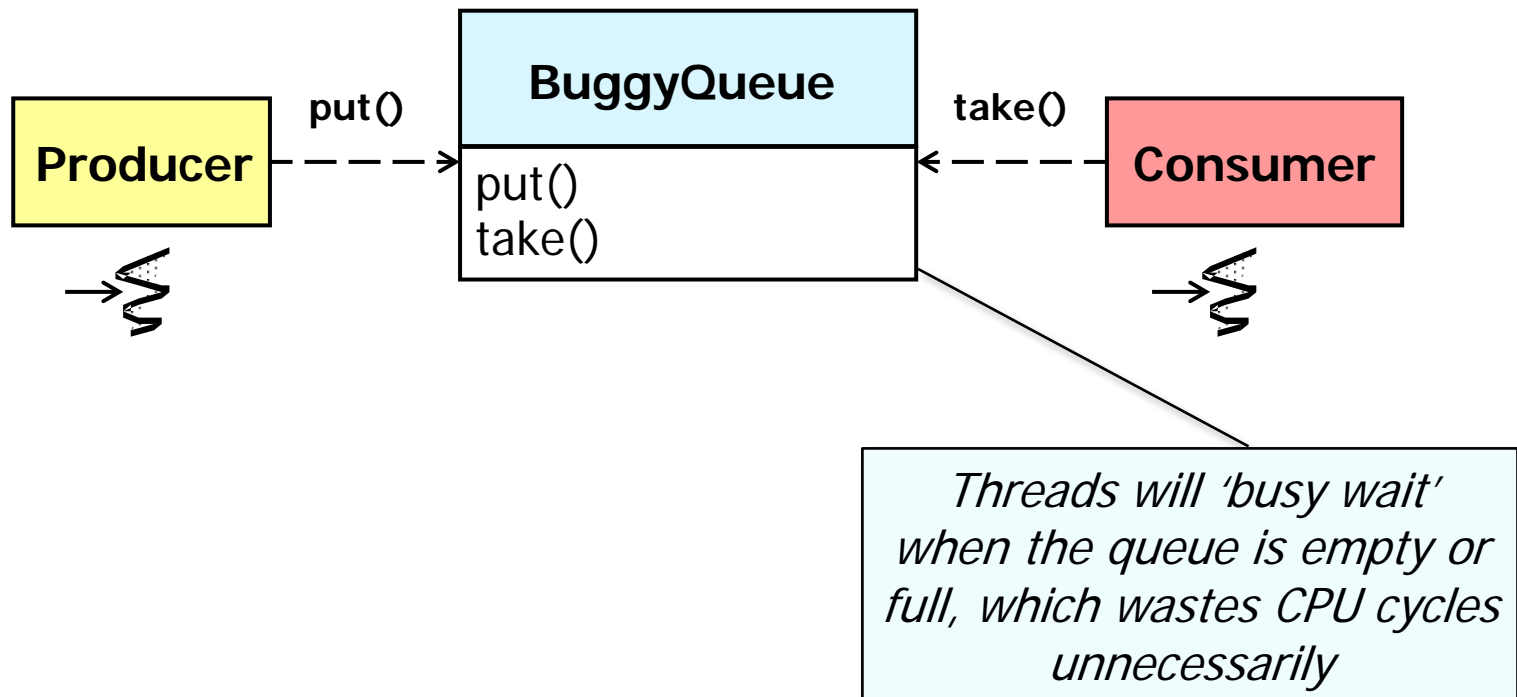
- Naïve implementations incur race conditions or “busy waiting” when multiple threads put/take items into/from the shared queue



# Challenge: Implementing a Blocking Queue

## Problems

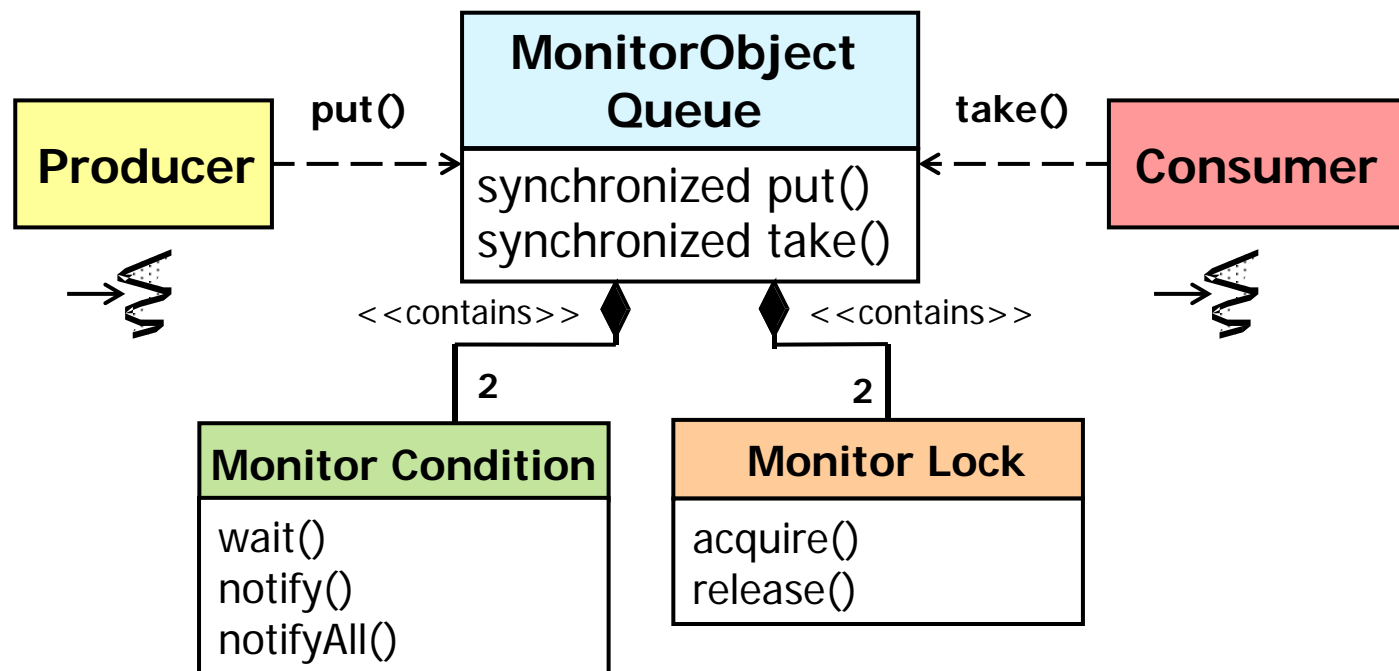
- Naïve implementations incur race conditions or “busy waiting” when multiple threads put/take items into/from the shared queue



# Challenge: Implementing a Blocking Queue

## Solution

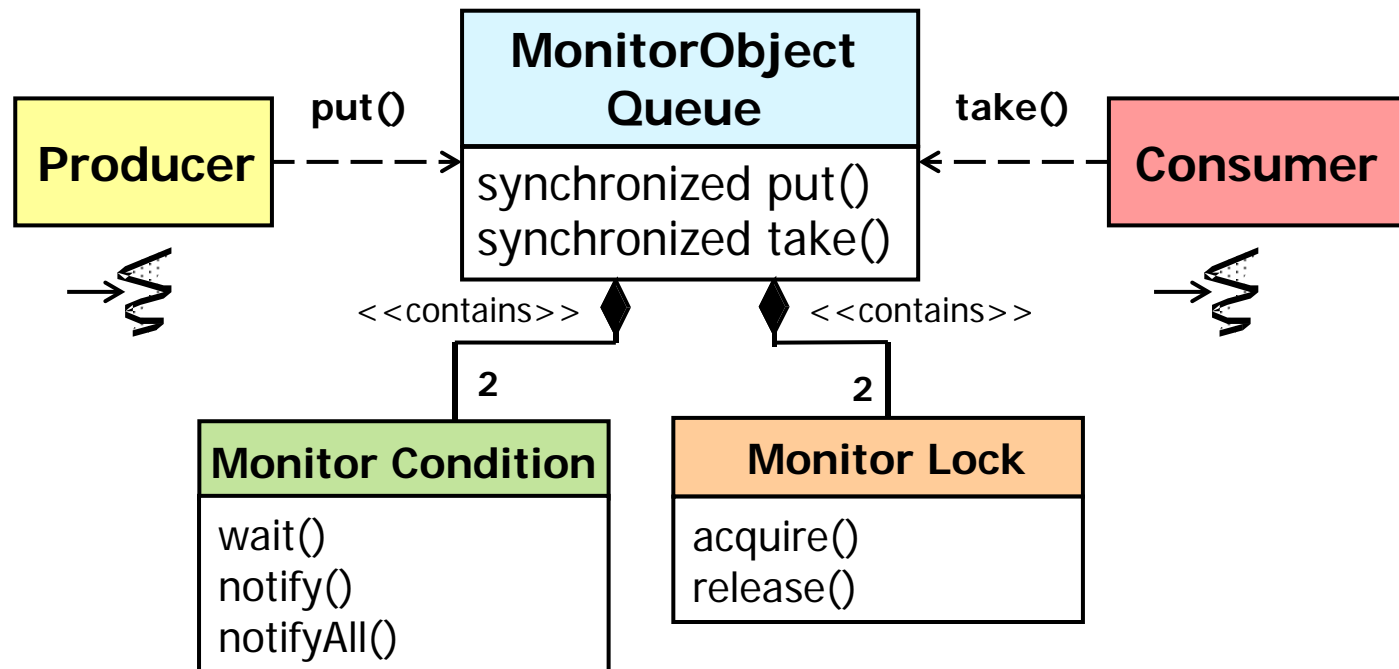
- Apply the *Monitor Object* pattern to synchronize the shared queue efficiently & conveniently



# Challenge: Implementing a Blocking Queue

## Solution

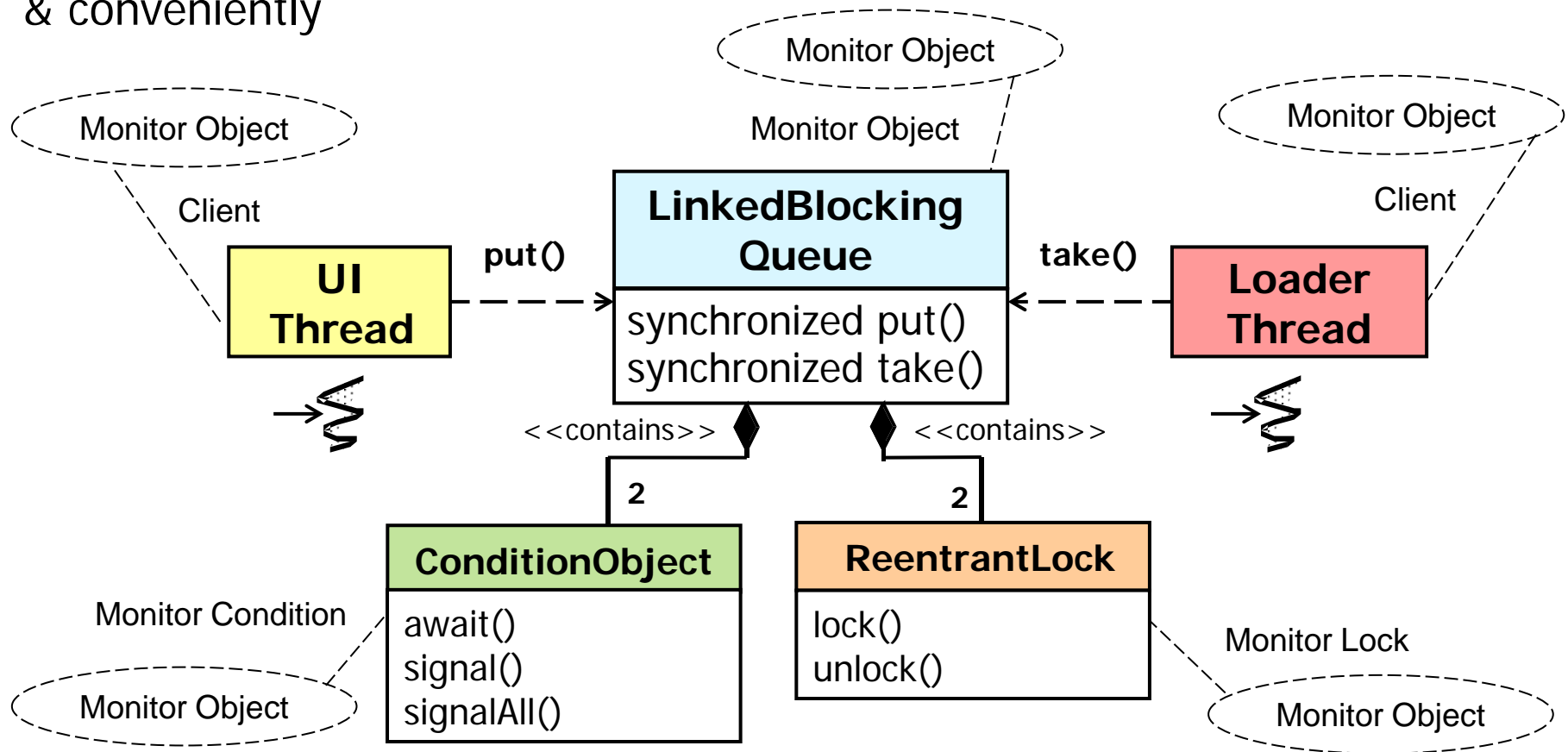
- Apply the *Monitor Object* pattern to synchronize the shared queue efficiently & conveniently



# Challenge: Implementing a Blocking Queue

## Solution

- Apply the *Monitor Object* pattern to synchronize the shared queue efficiently & conveniently

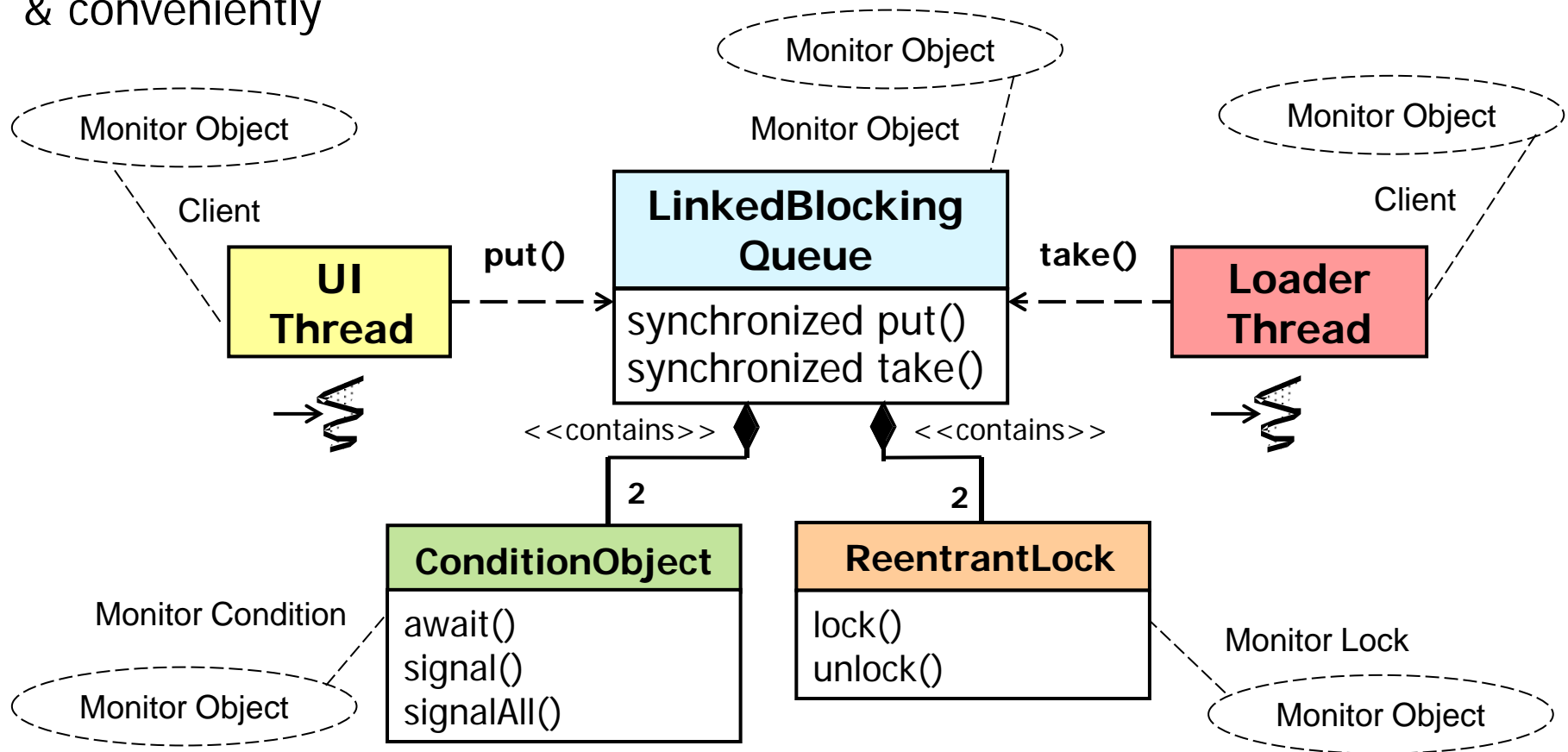




# Challenge: Implementing a Blocking Queue

## Solution

- Apply the *Monitor Object* pattern to synchronize the shared queue efficiently & conveniently



See earlier part on "Java ConditionObject" for ArrayBlockingQueue analysis

# Challenge: Implementing a Blocking Queue

## Solution

- Apply the *Monitor Object* pattern to synchronize the shared queue efficiently & conveniently

### LinkedBlocking Queue

synchronized put()  
synchronized take()



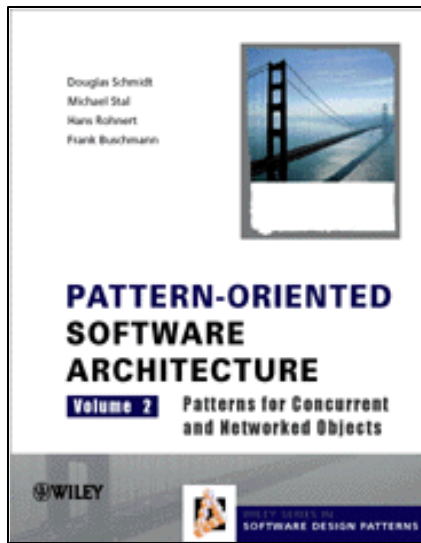
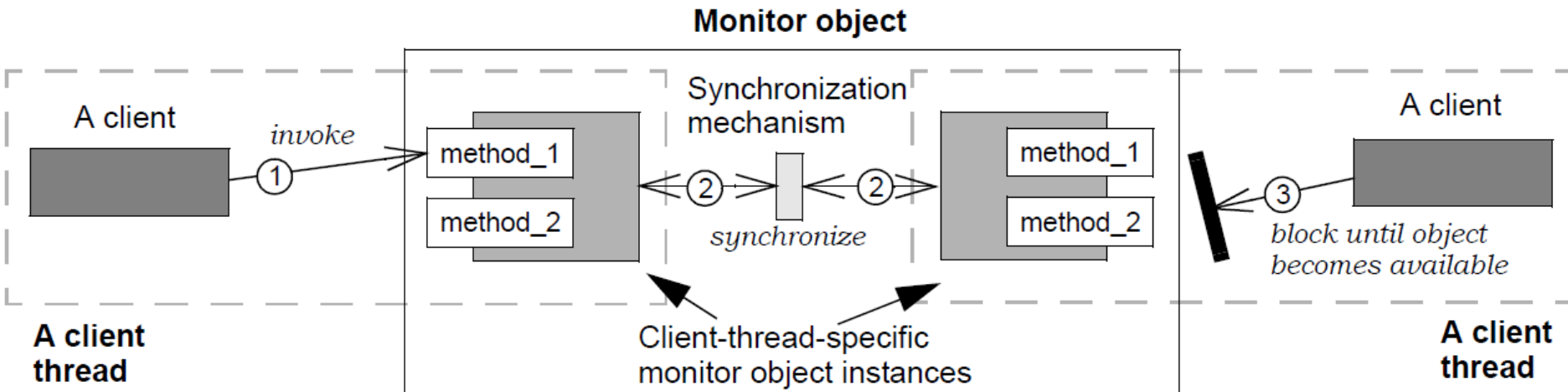
### ArrayBlock Queue

synchronized put()  
synchronized take()

# Intent & Applicability of the Monitor Object Pattern

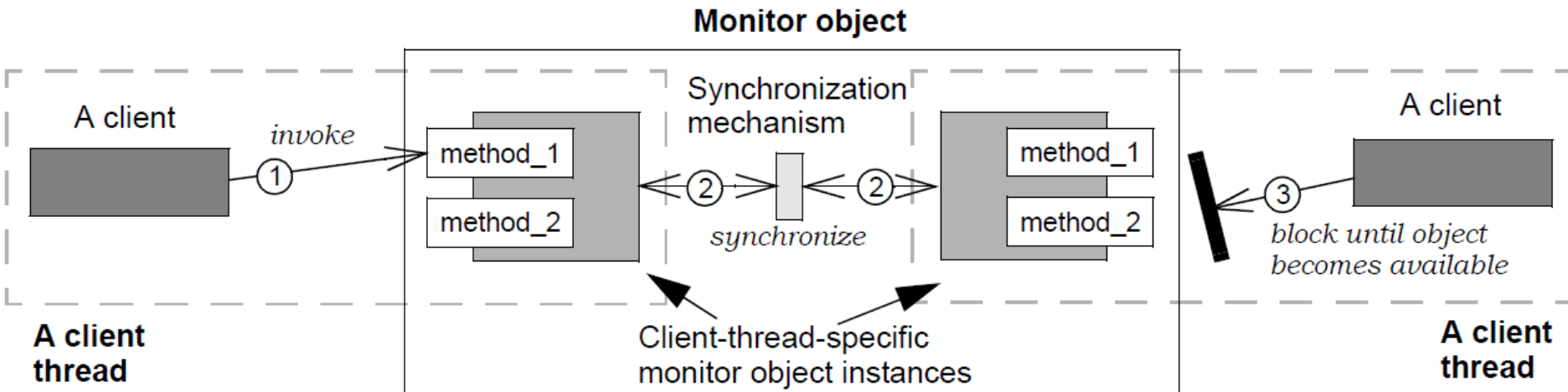
# Monitor Object

# POSA2 Concurrency



# Monitor Object

# POSA2 Concurrency

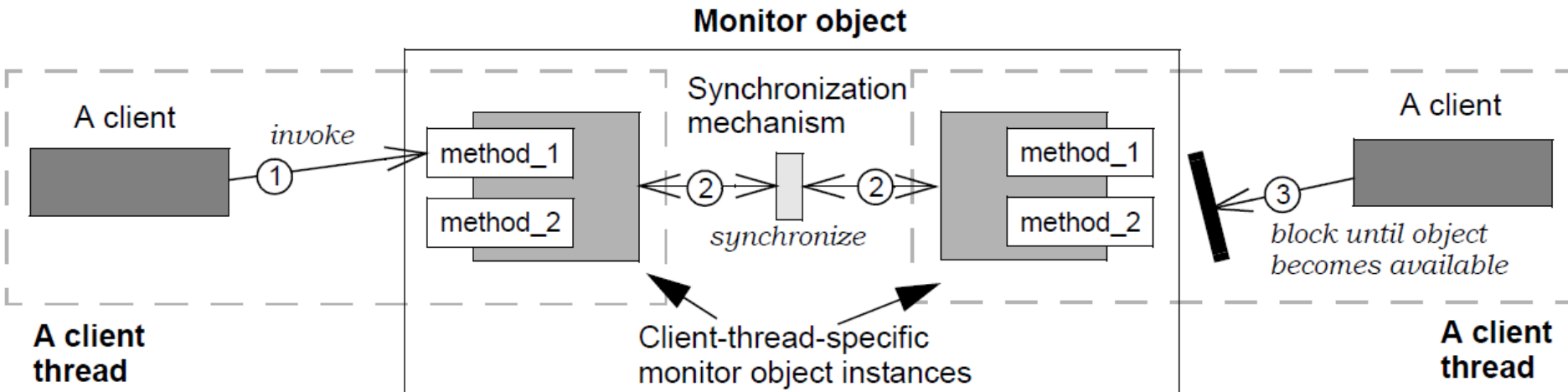


## Intent

- Synchronizes concurrent method execution to ensure only one method at a time runs within an object

# Monitor Object

# POSA2 Concurrency

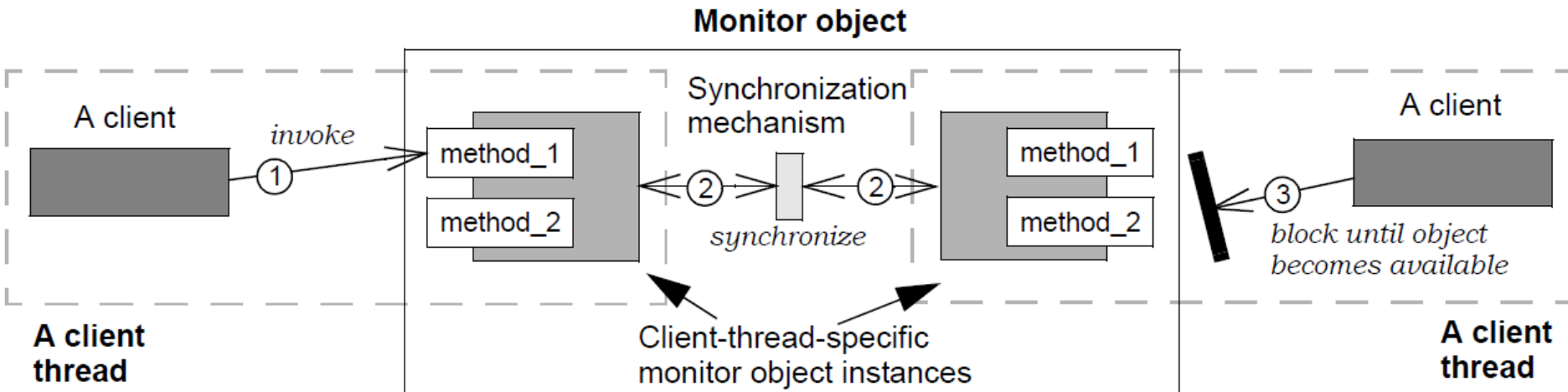


## Intent

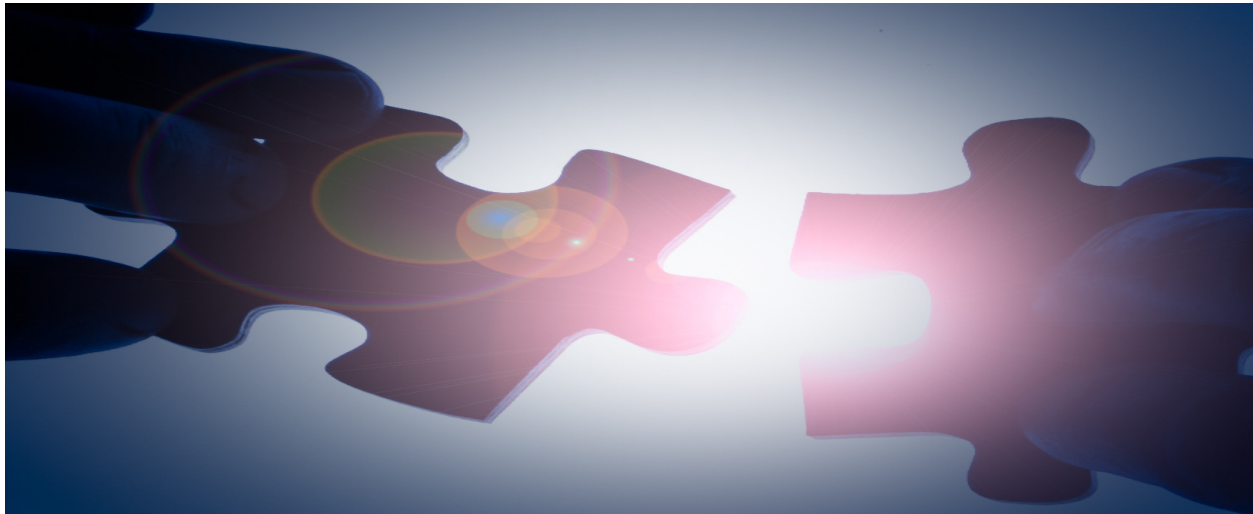
- Synchronizes concurrent method execution to ensure only one method at a time runs within an object
- Allows an object's methods to cooperatively schedule their execution sequences

# Monitor Object

# POSA2 Concurrency

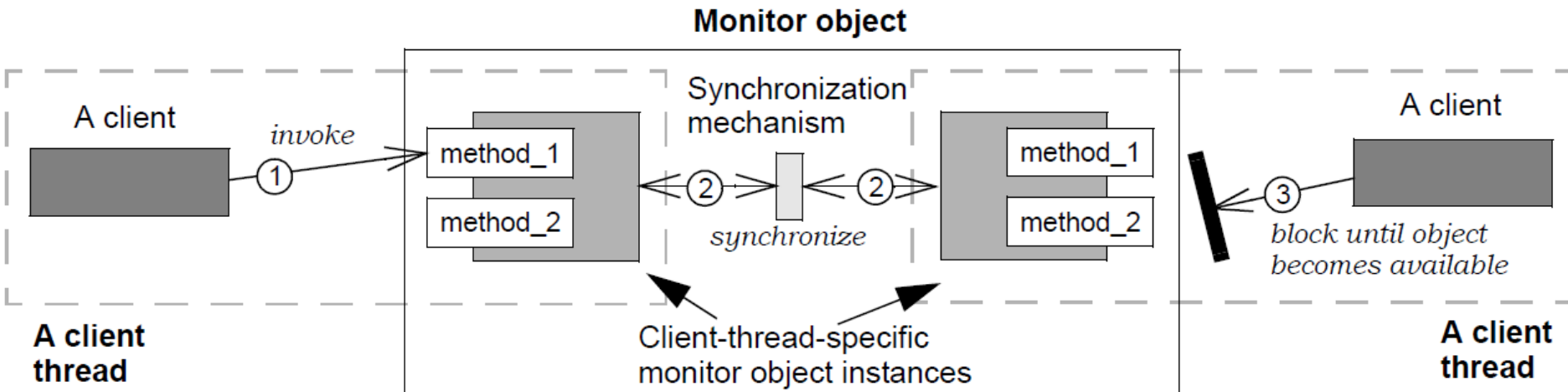


## Applicability



# Monitor Object

# POSA2 Concurrency



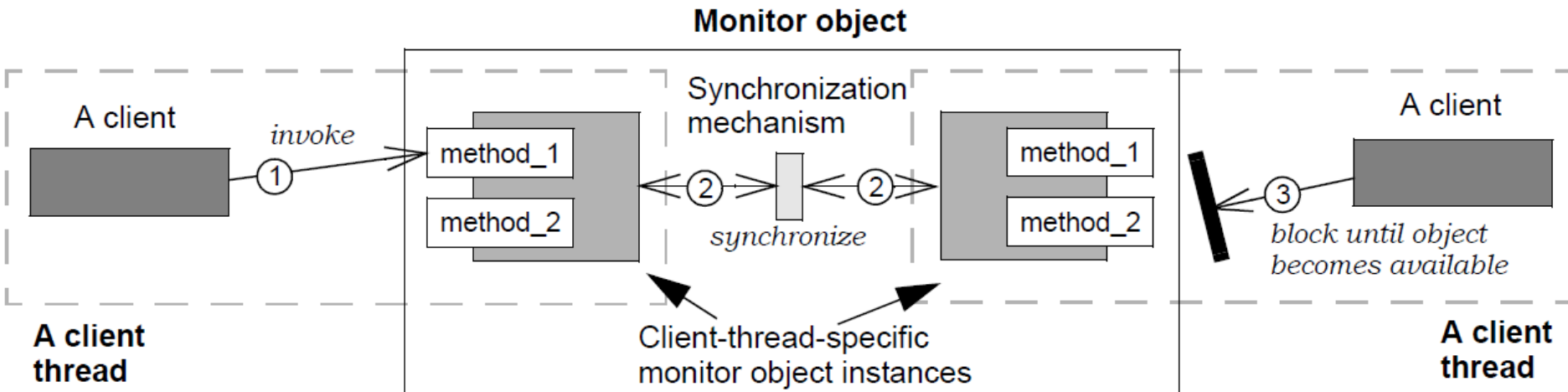
## Applicability

- When an object's interface methods can define its synchronization & scheduling boundaries



# Monitor Object

# POSA2 Concurrency

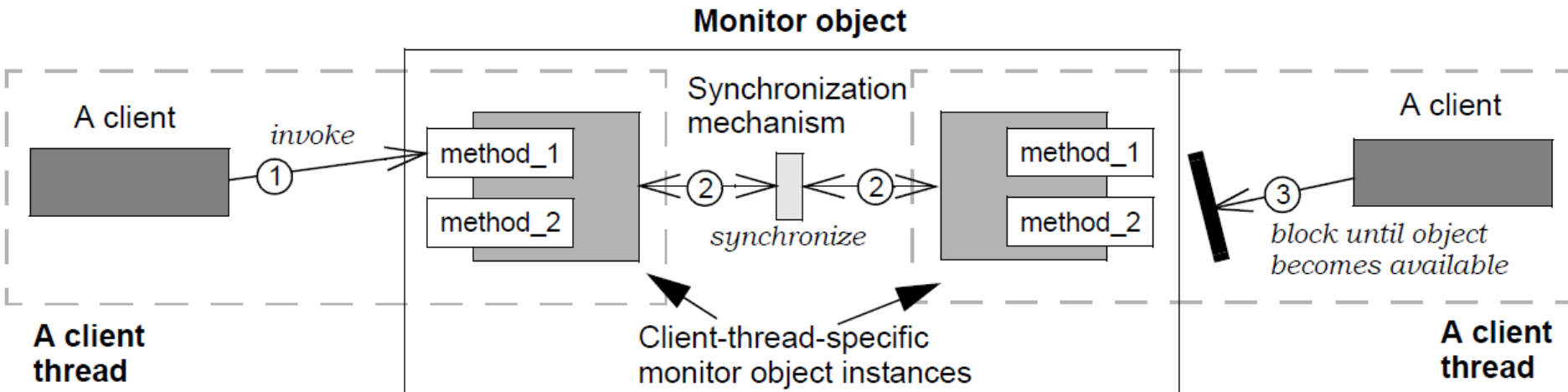


## Applicability

- When an object's interface methods can define its synchronization & scheduling boundaries
- This is an extension of the traditional object-oriented programming model

# Monitor Object

# POSA2 Concurrency

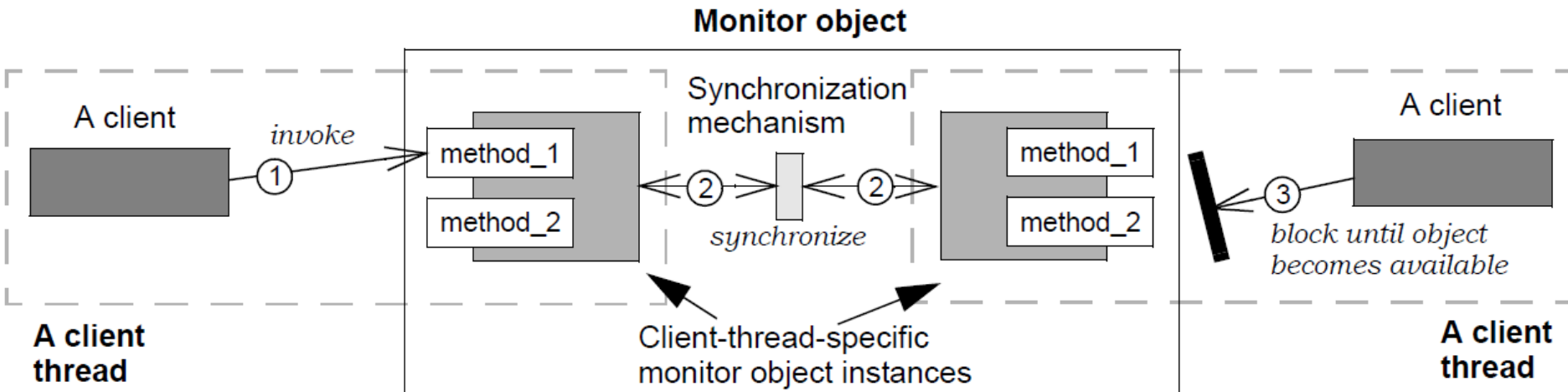


## Applicability

- When an object's interface methods can define its synchronization & scheduling boundaries
- When only one method at a time should be active within an object

# Monitor Object

# POSA2 Concurrency

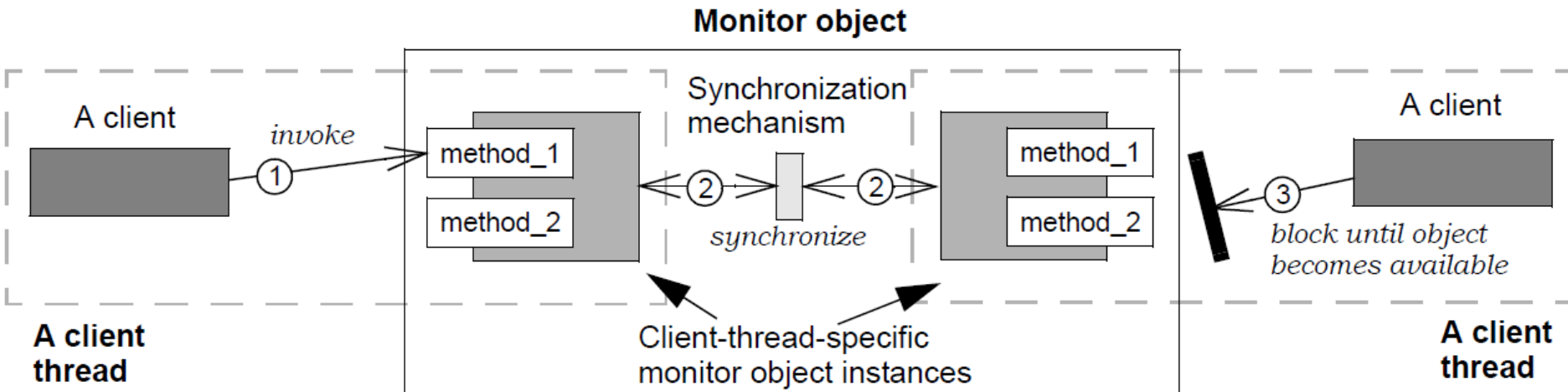


## Applicability

- When an object's interface methods can define its synchronization & scheduling boundaries
- When only one method at a time should be active within an object
- When objects should be responsible for transparent method serialization

# Monitor Object

# POSA2 Concurrency

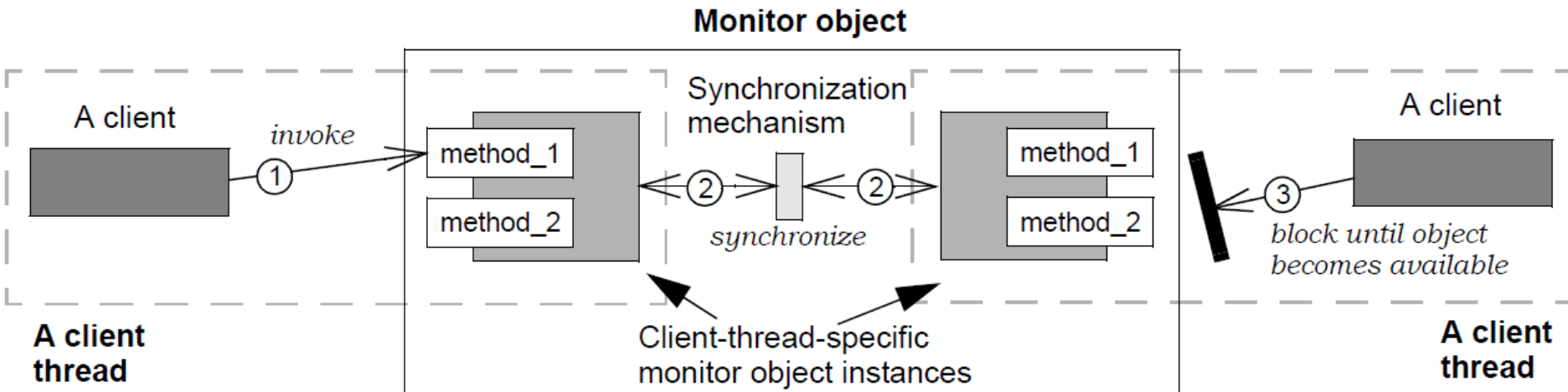


## Applicability

- When an object's interface methods can define its synchronization & scheduling boundaries
- When only one method at a time should be active within an object
- When objects should be responsible for transparent method serialization
  - It's tedious & error-prone for clients to explicitly acquire & release low-level synchronization & scheduling mechanisms

# Monitor Object

# POSA2 Concurrency

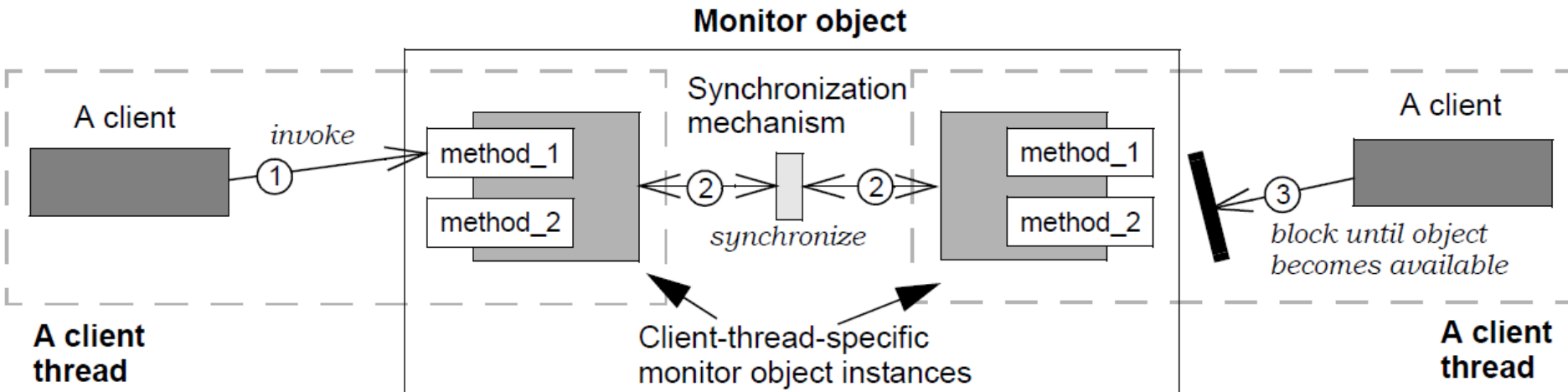


## Applicability

- When an object's interface methods can define its synchronization & scheduling boundaries
- When only one method at a time should be active within an object
- When objects should be responsible for transparent method serialization
- When an object's methods interact cooperatively via multiple threads

# Monitor Object

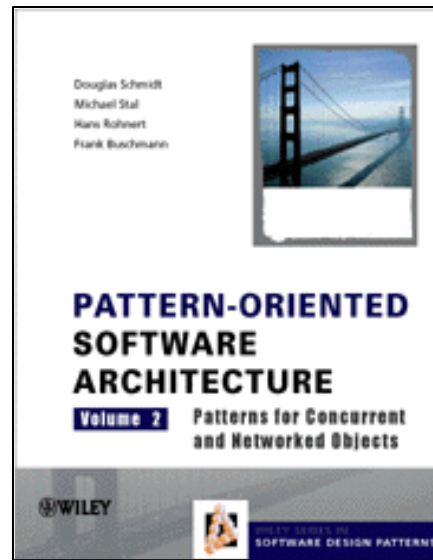
# POSA2 Concurrency



## Applicability

- When an object's interface methods can define its synchronization & scheduling boundaries
- When only one method at a time should be active within an object
- When objects should be responsible for transparent method serialization
- When an object's methods interact cooperatively via multiple threads
  - Object-specific invariants must hold as threads suspend & resume their execution

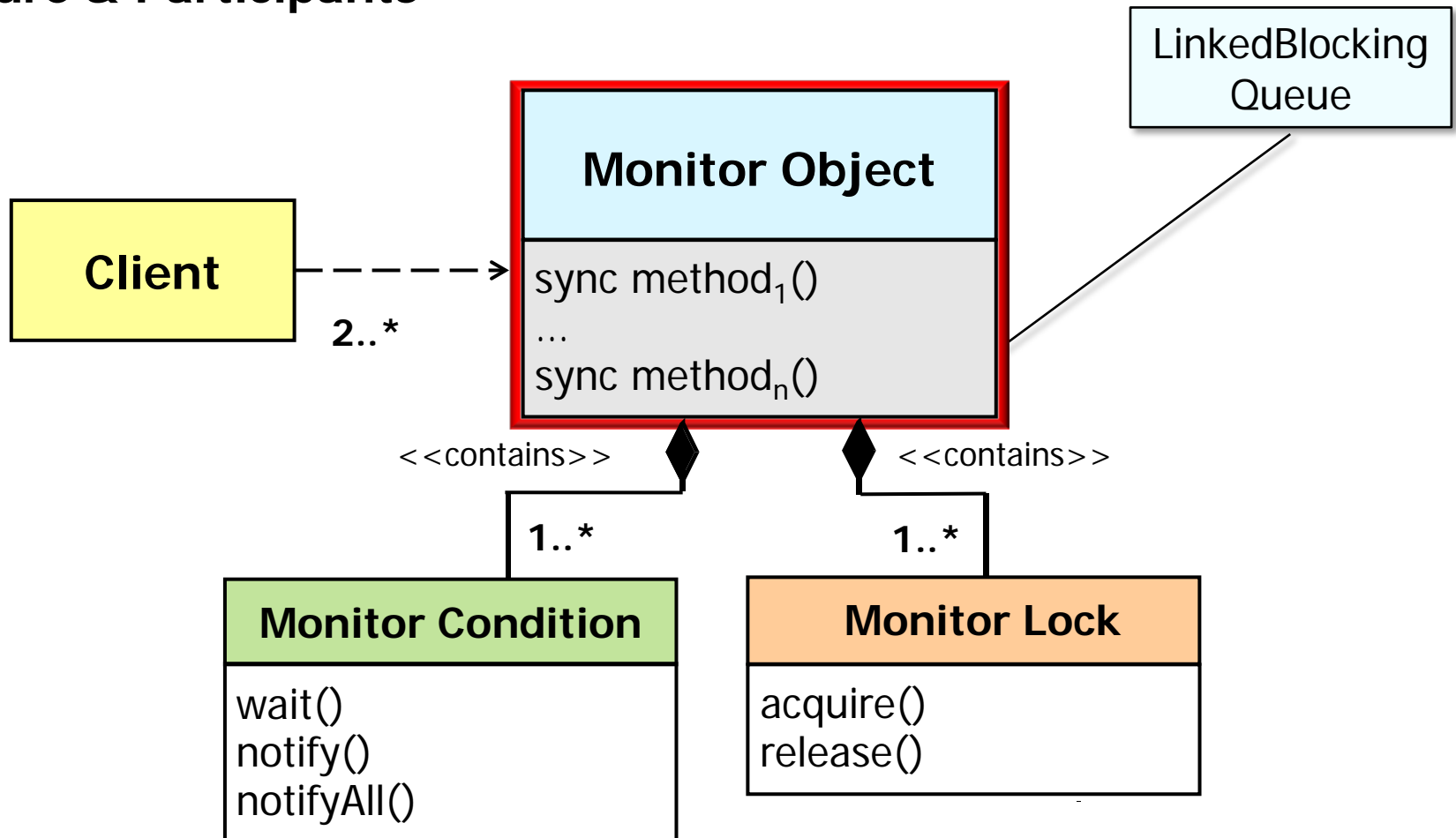
# Structure of the Monitor Object Pattern



# Monitor Object

# POSA2 Concurrency

## Structure & Participants

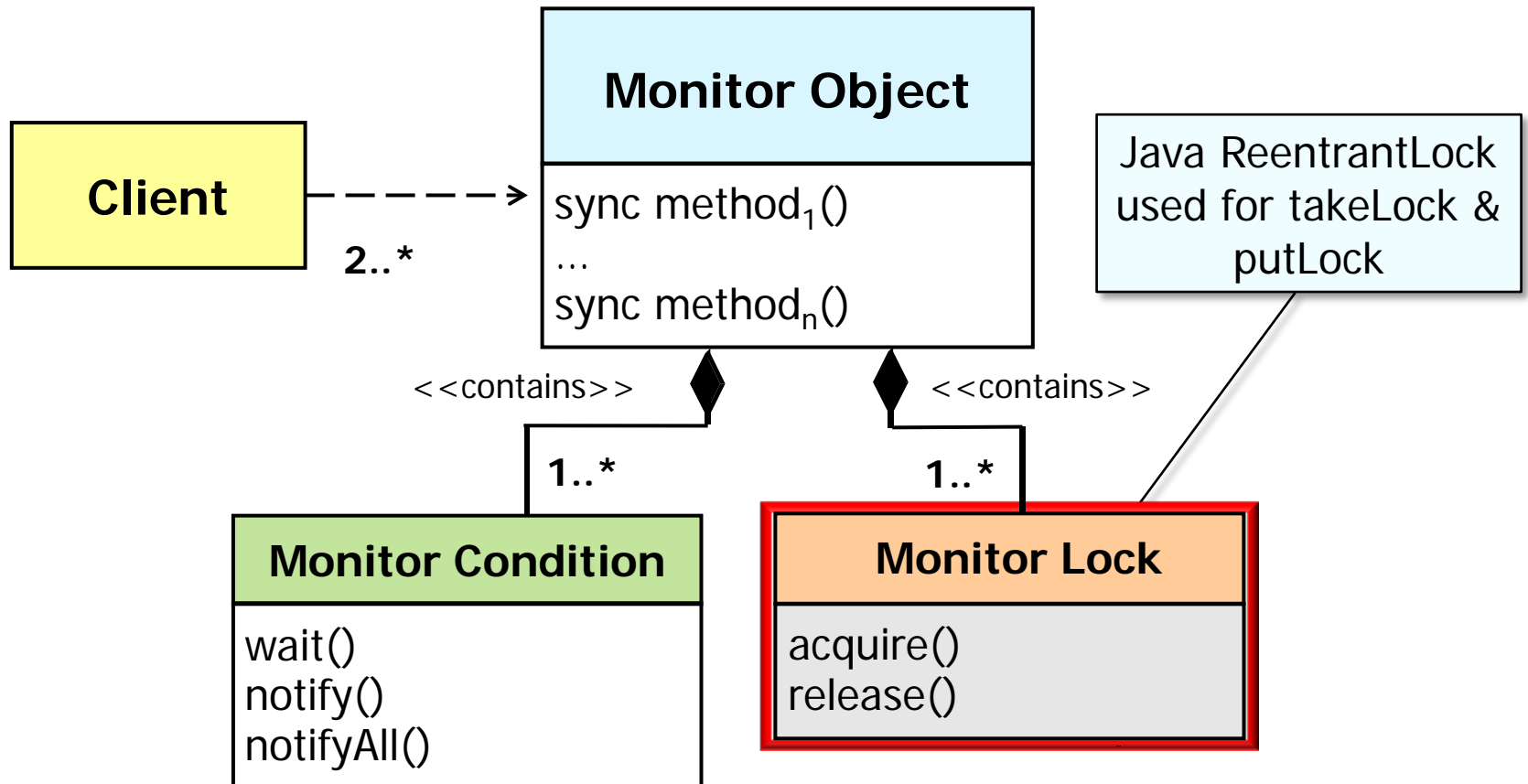




# Monitor Object

# POSA2 Concurrency

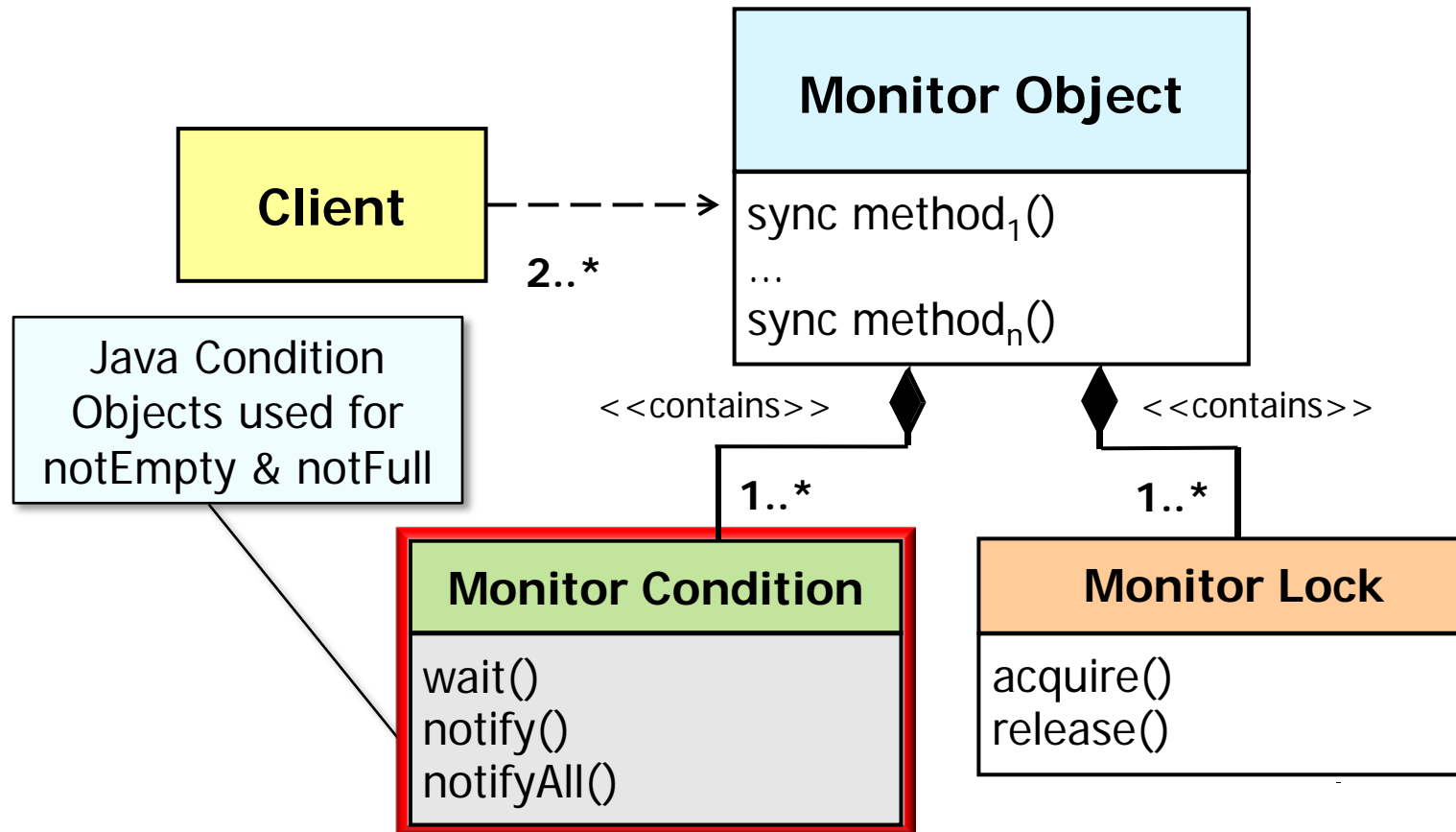
## Structure & Participants



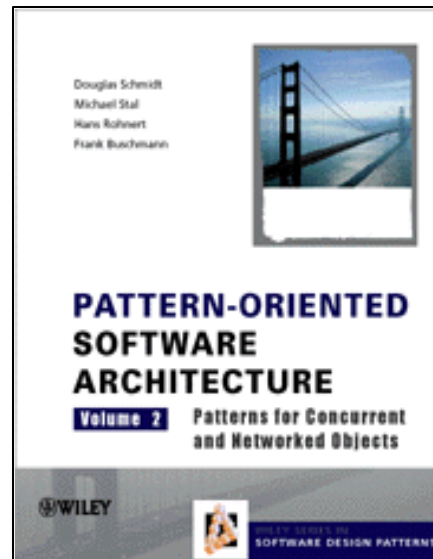
# Monitor Object

# POSA2 Concurrency

## Structure & Participants



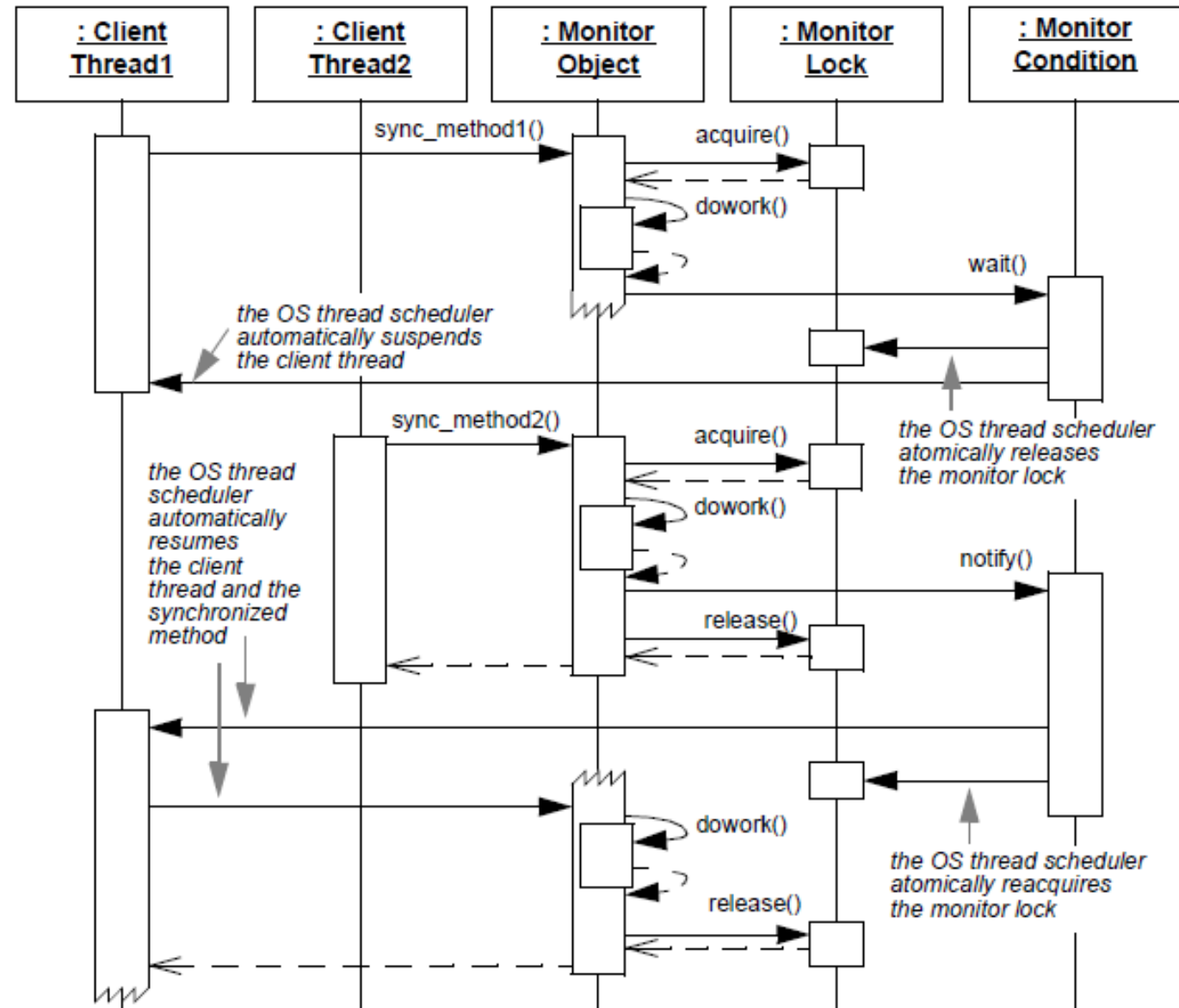
# Dynamics of the Monitor Object Pattern



# Monitor Object

# POSA2 Concurrency

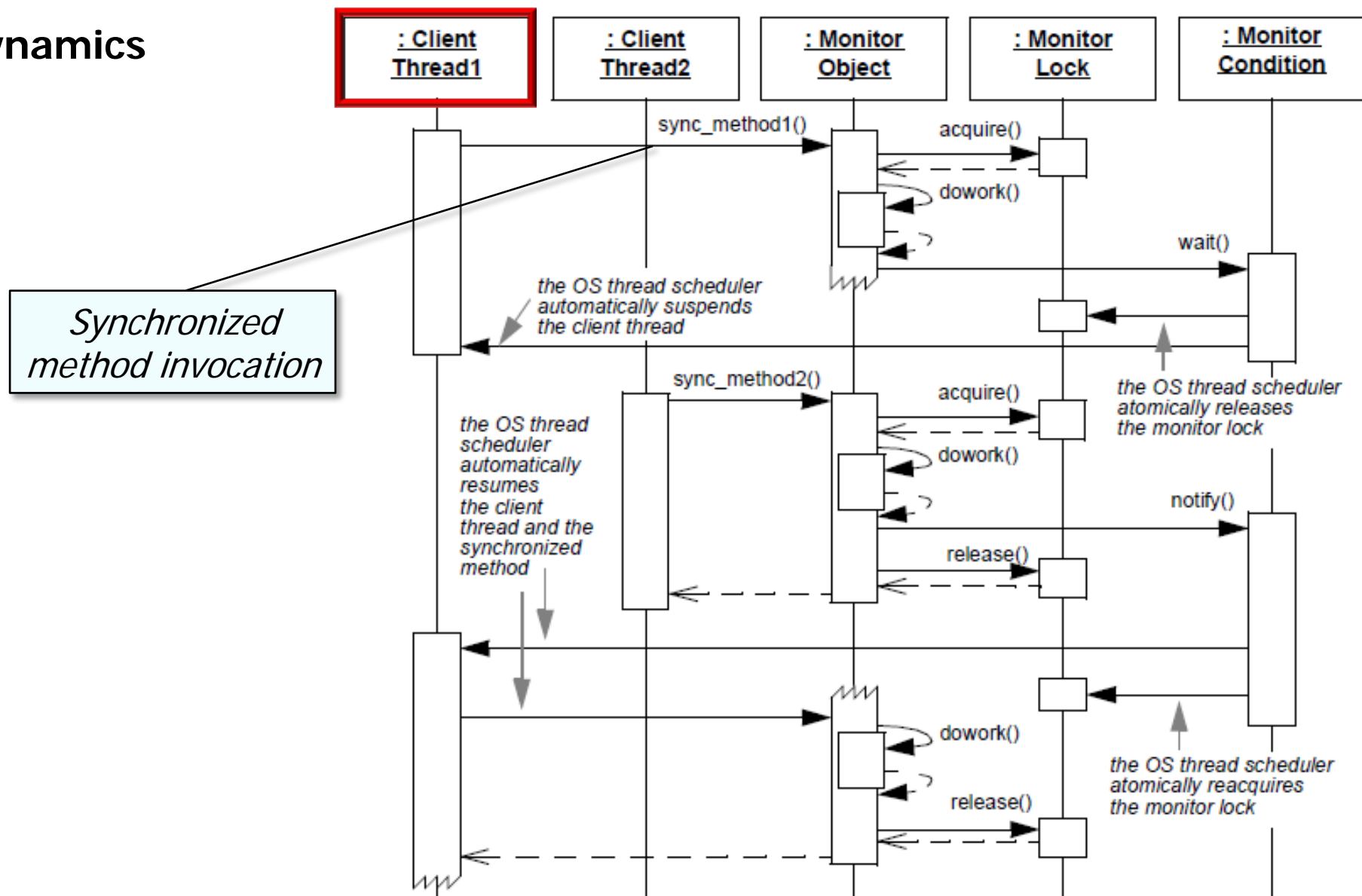
## Dynamics



# Monitor Object

# POSA2 Concurrency

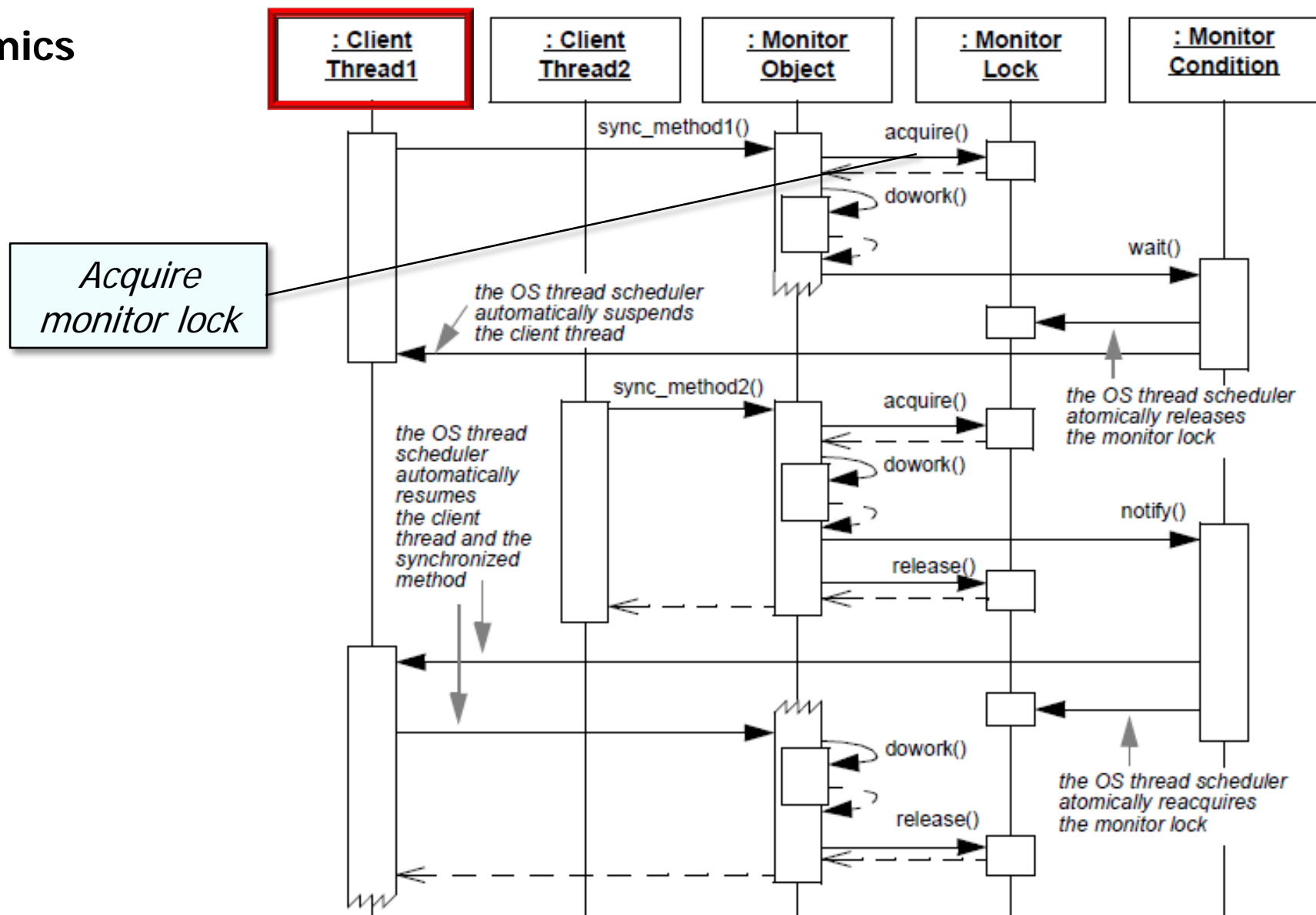
## Dynamics



# Monitor Object

# POSA2 Concurrency

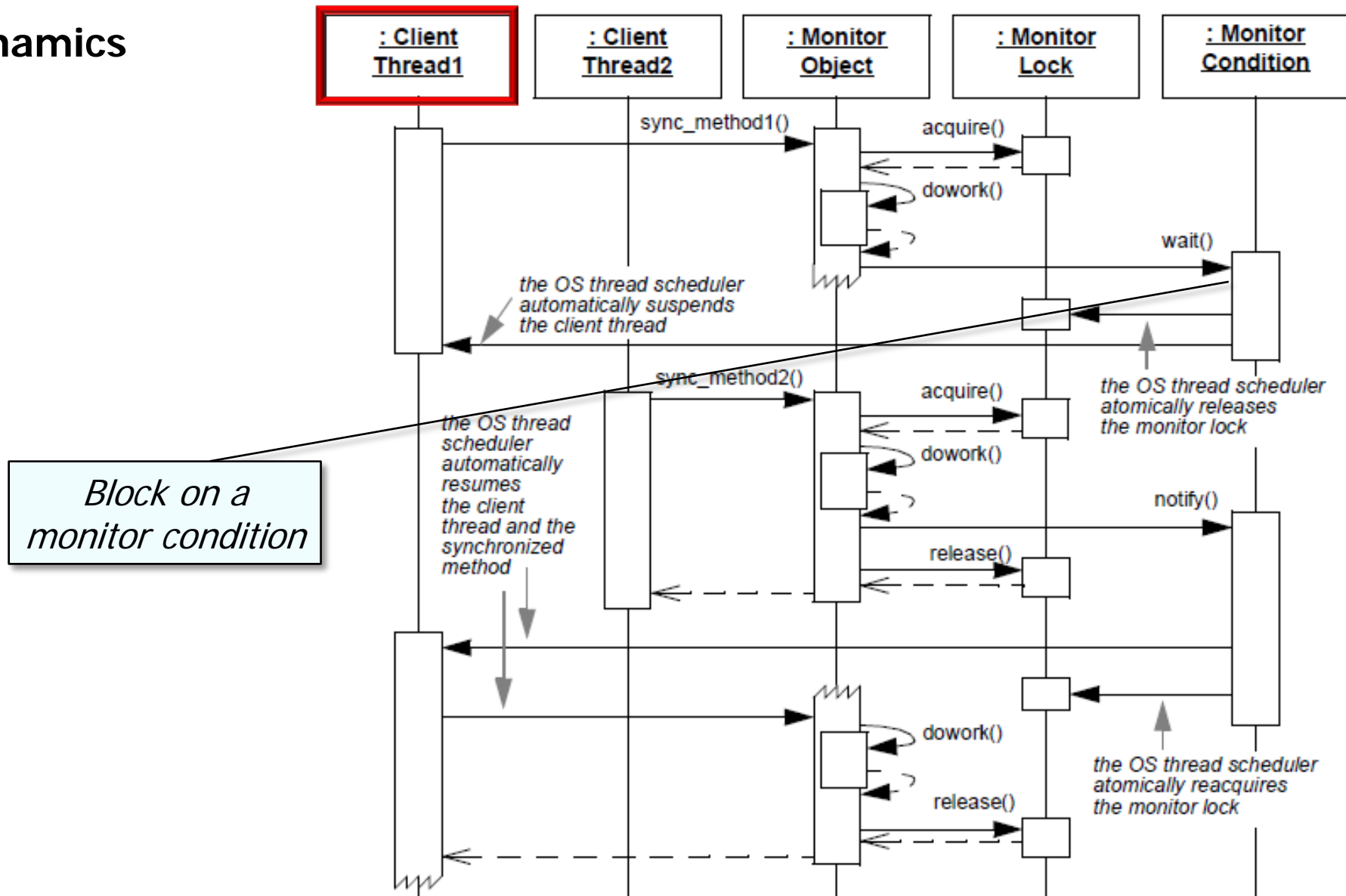
## Dynamics



# Monitor Object

# POSA2 Concurrency

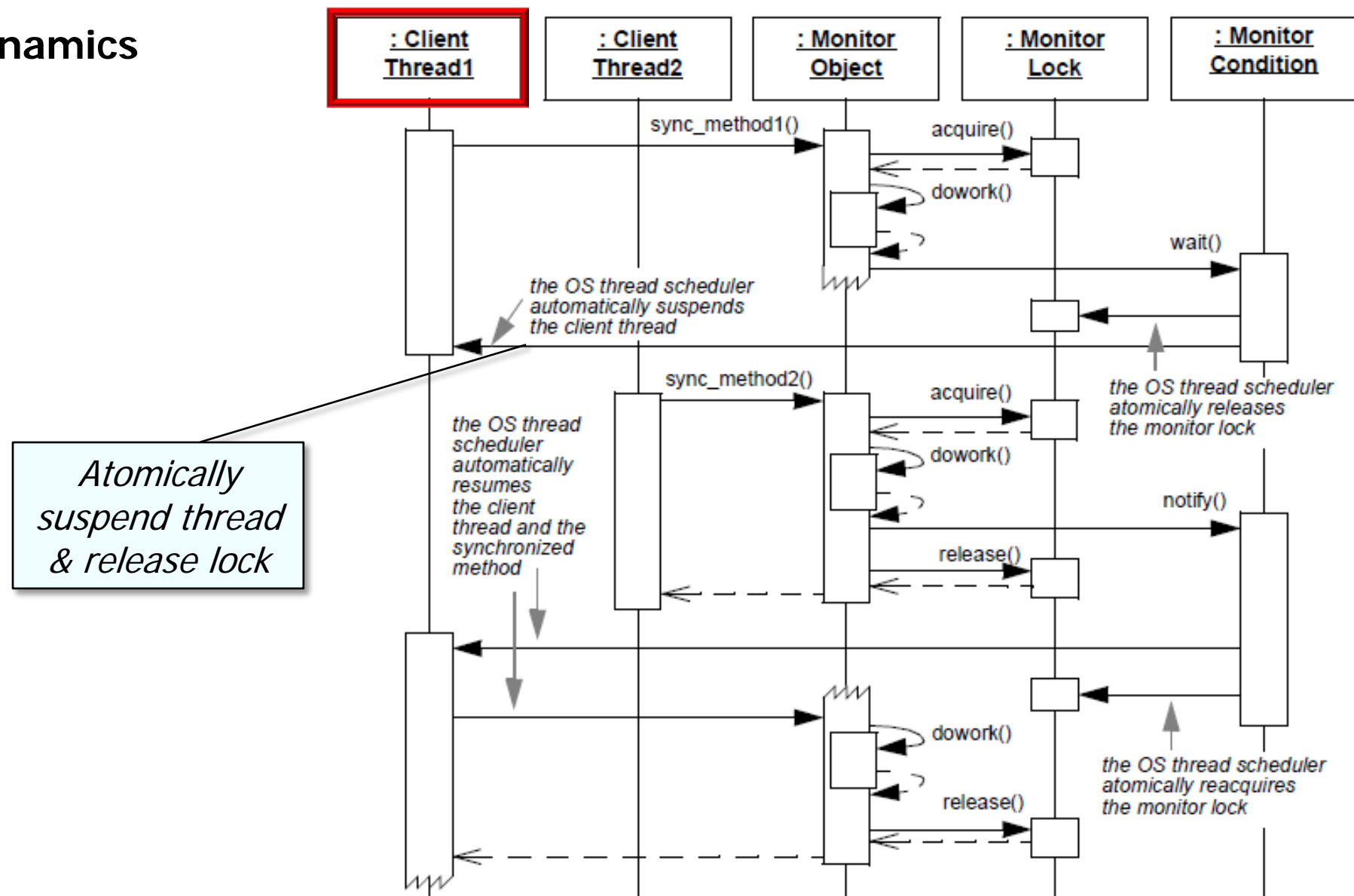
## Dynamics



# Monitor Object

# POSA2 Concurrency

## Dynamics



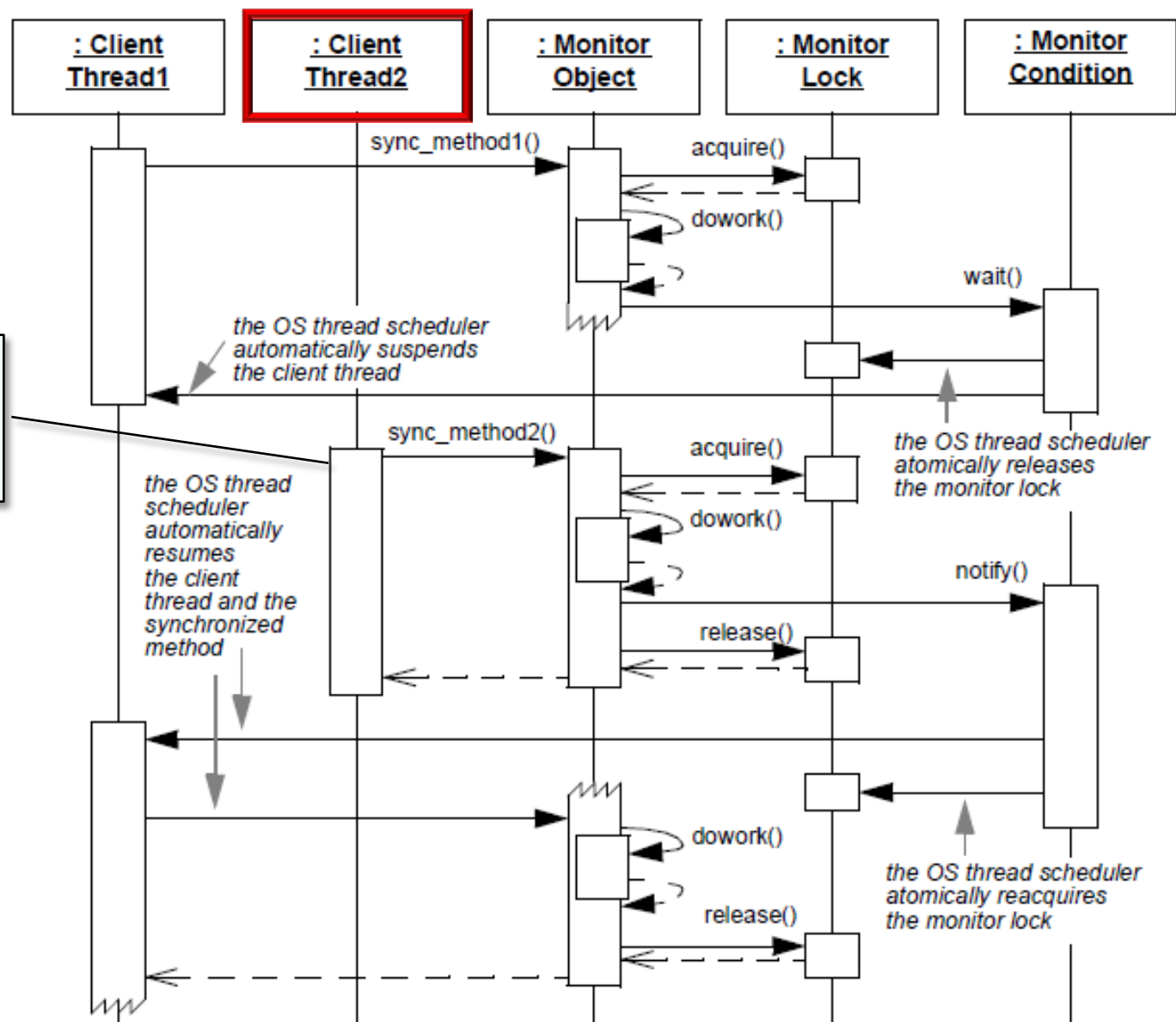


# Monitor Object

# POSA2 Concurrency

## Dynamics

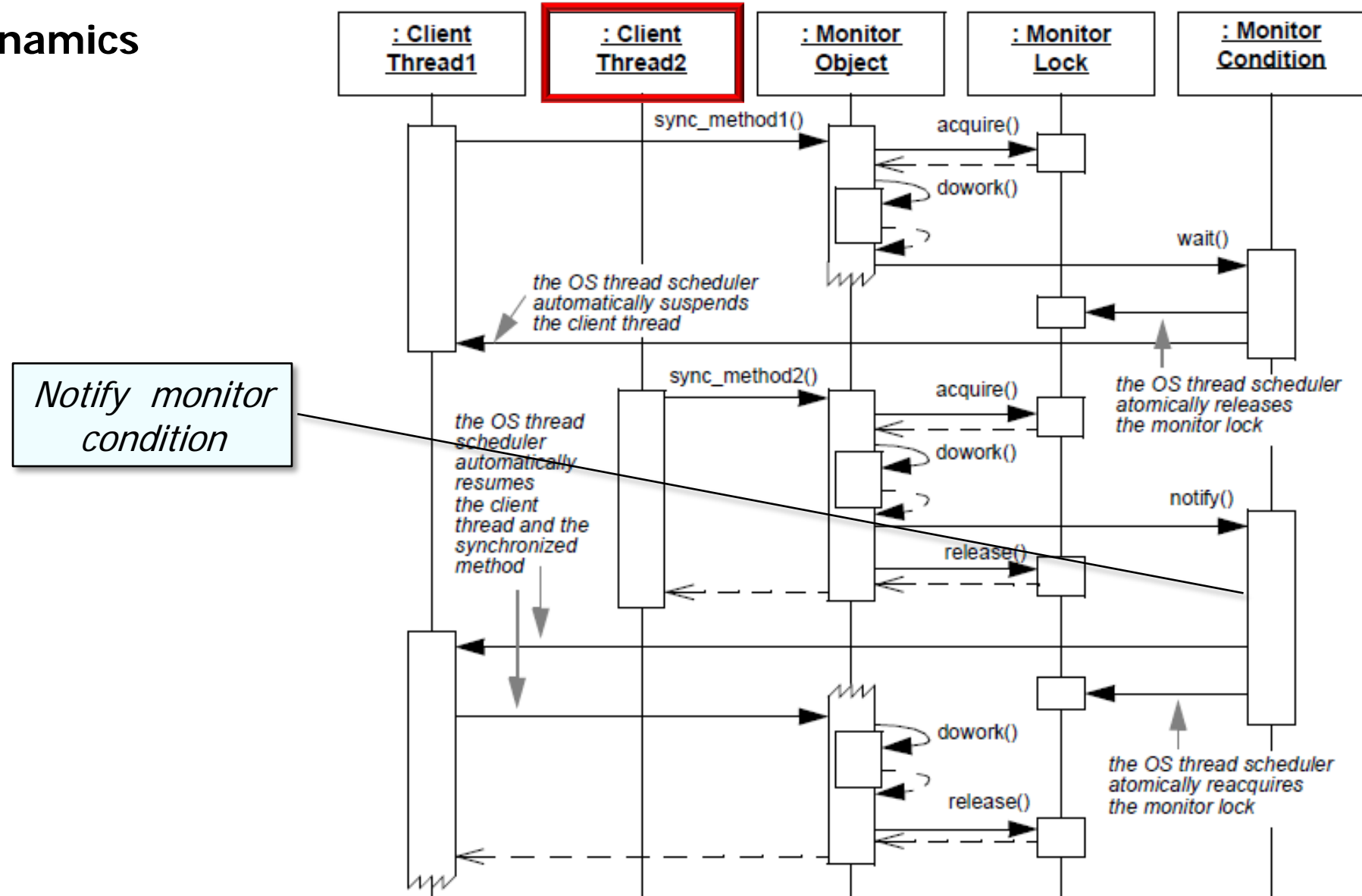
*Allow another thread to run in the monitor object*



# Monitor Object

# POSA2 Concurrency

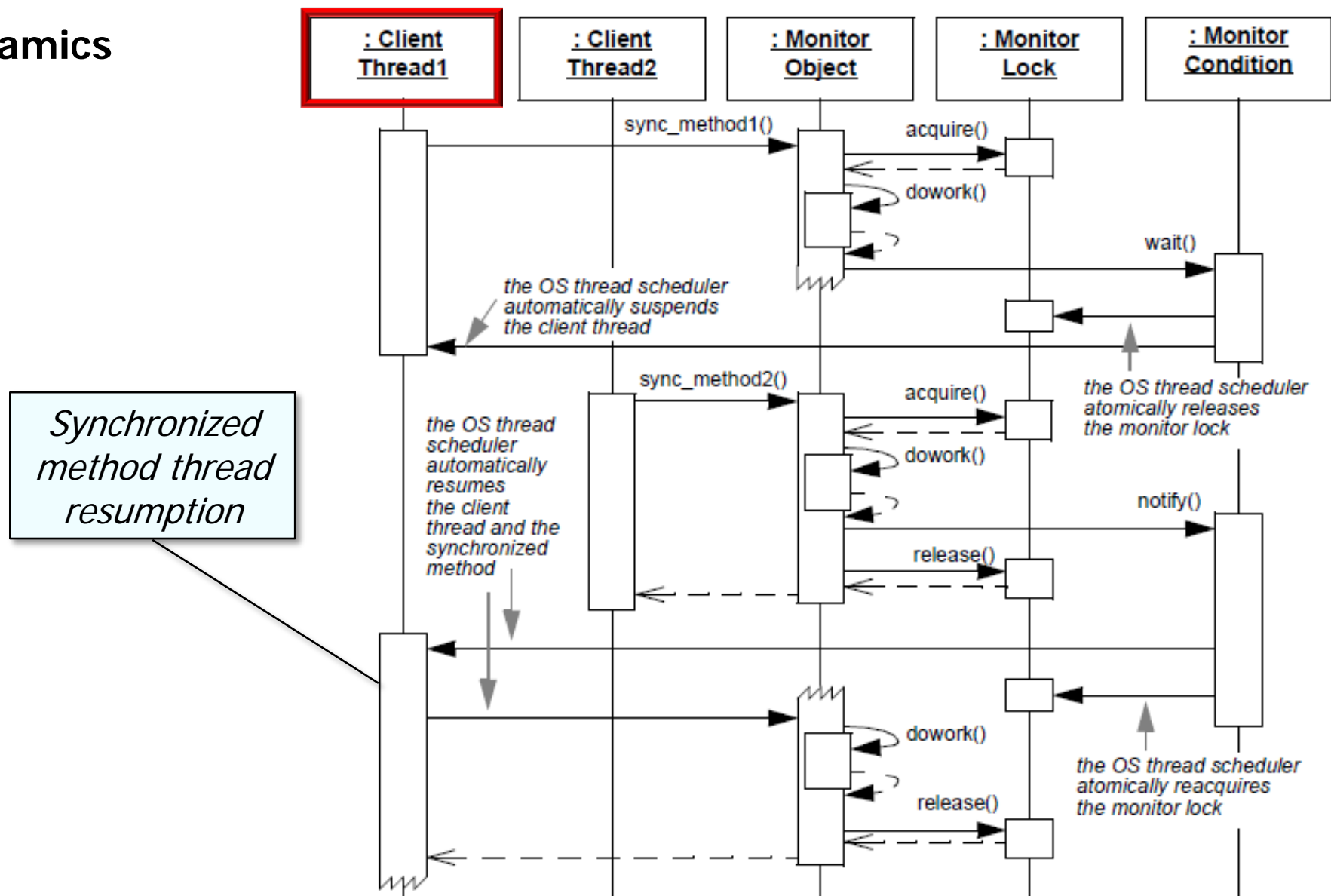
## Dynamics



# Monitor Object

# POSA2 Concurrency

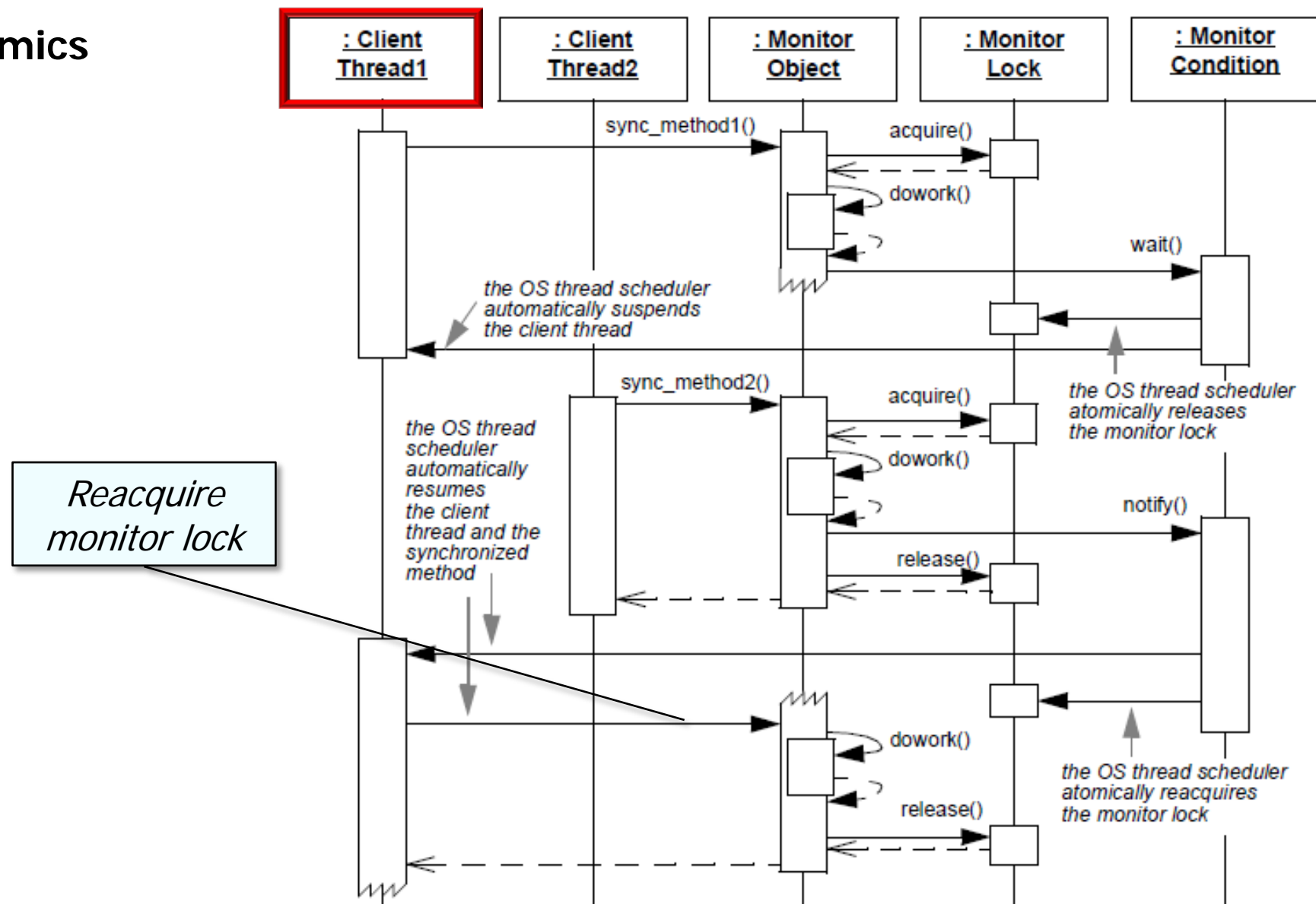
## Dynamics



# Monitor Object

# POSA2 Concurrency

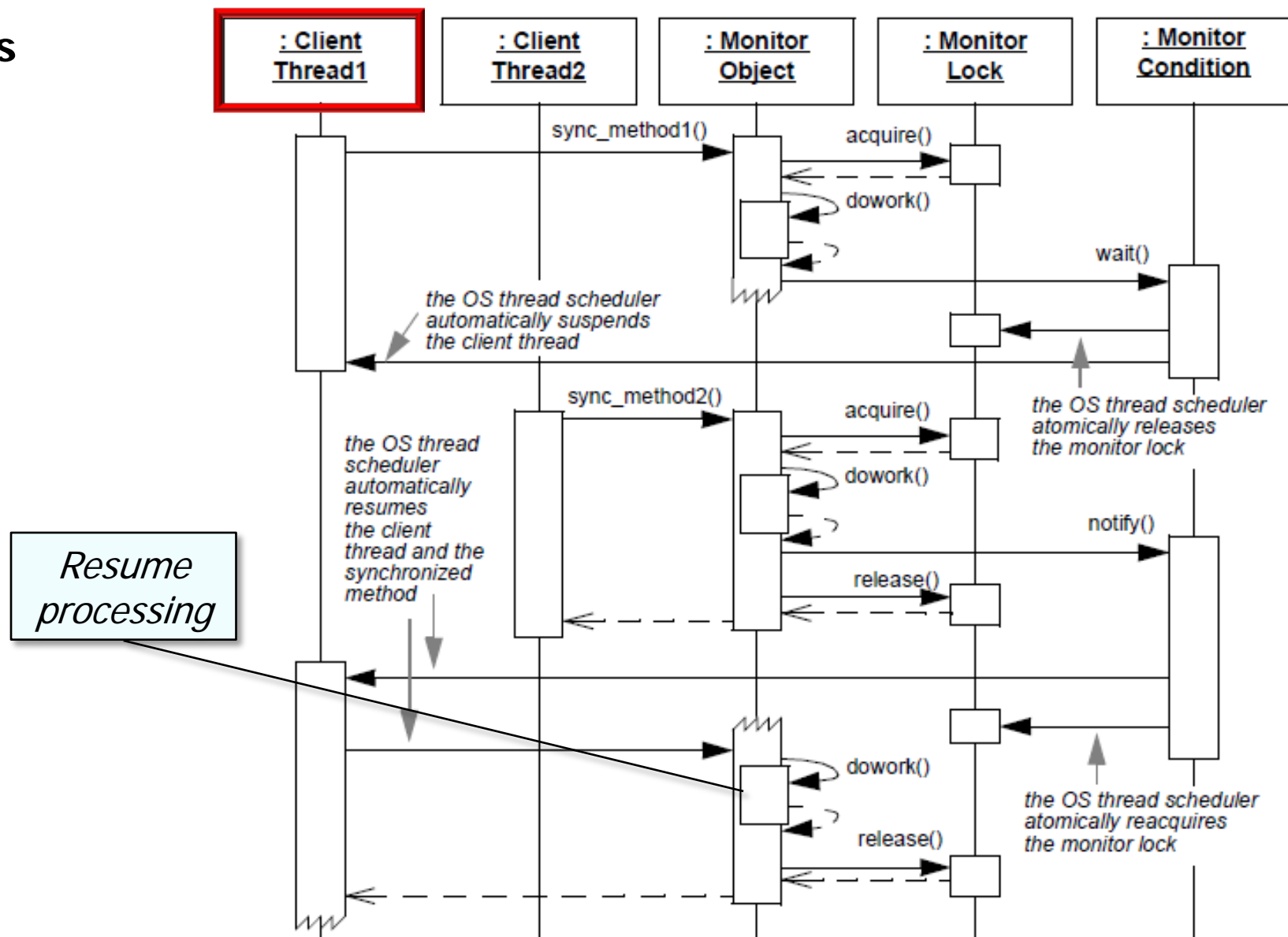
## Dynamics



# Monitor Object

# POSA2 Concurrency

## Dynamics



# Consequences of the Monitor Object Pattern

# Monitor Object

# POSA2 Concurrency

**Consequences**

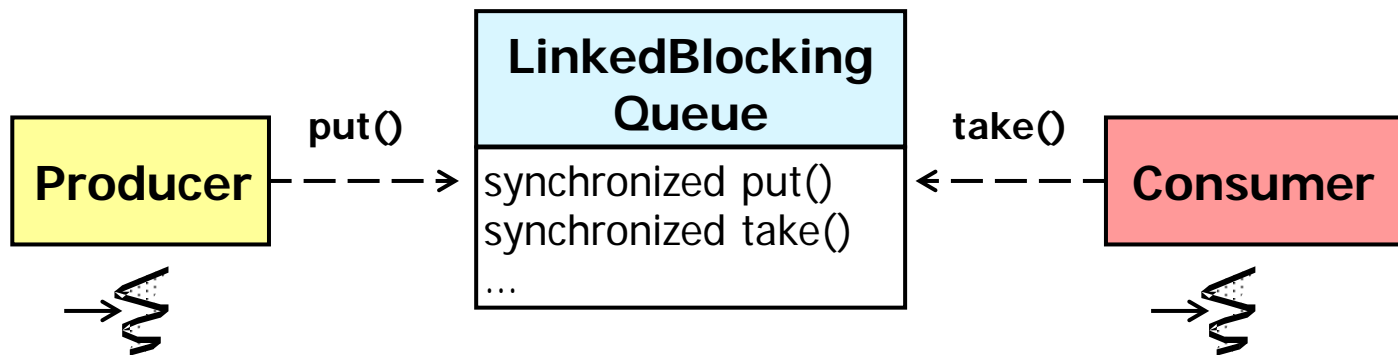


# Monitor Object

# POSA2 Concurrency

## Consequences

- + Concise programming model for concurrency control
  - Simplifies sharing an object among cooperating threads by aligning synchronization transparently with method invocations



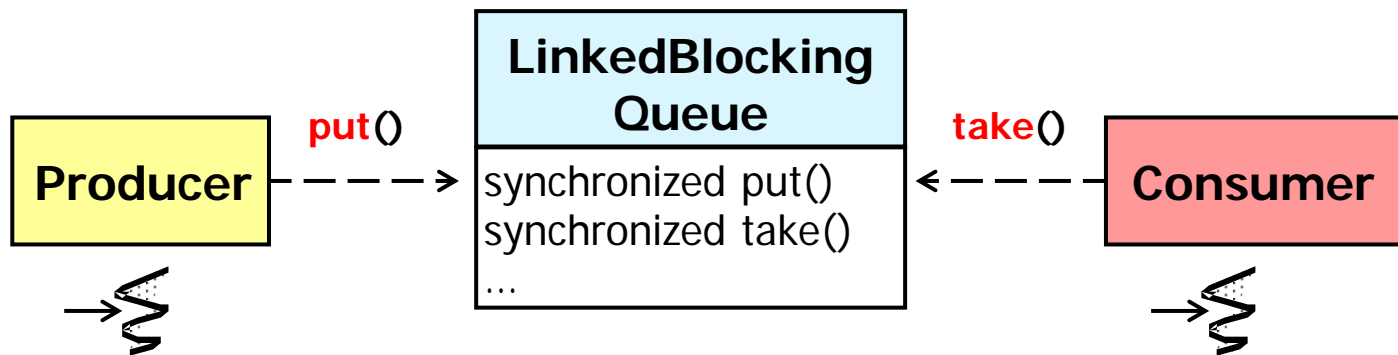


# Monitor Object

# POSA2 Concurrency

## Consequences

- + Concise programming model for concurrency control
  - Simplifies sharing an object among cooperating threads by aligning synchronization transparently with method invocations

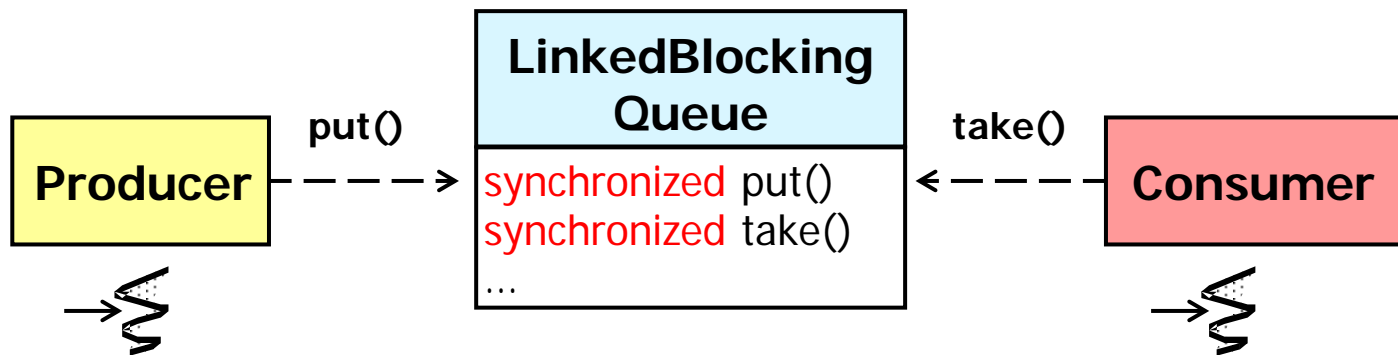


# Monitor Object

# POSA2 Concurrency

## Consequences

- + Concise programming model for concurrency control
  - Simplifies sharing an object among cooperating threads by aligning synchronization transparently with method invocations

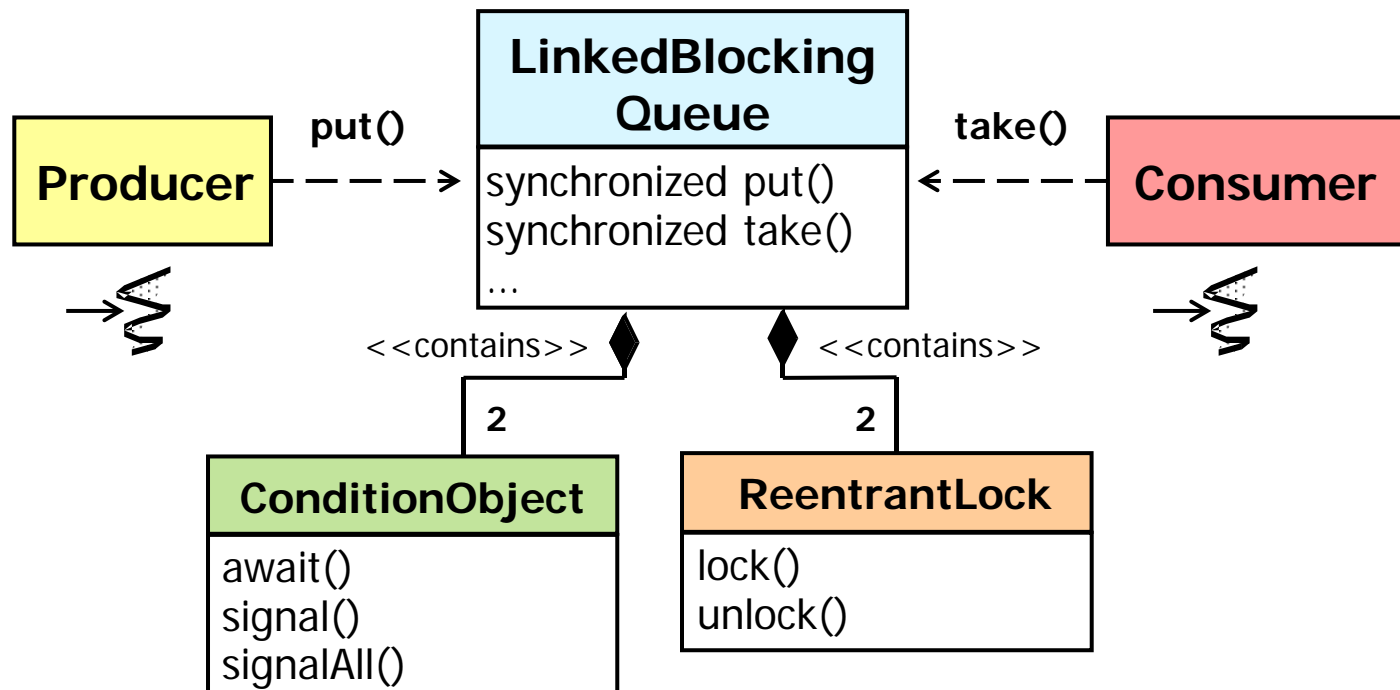


# Monitor Object

# POSA2 Concurrency

## Consequences

- + Concise programming model for concurrency control
- + Simplification of scheduling method execution
  - Synchronized methods use monitor conditions to determine when a thread should suspend or resume its execution & that of collaborating threads

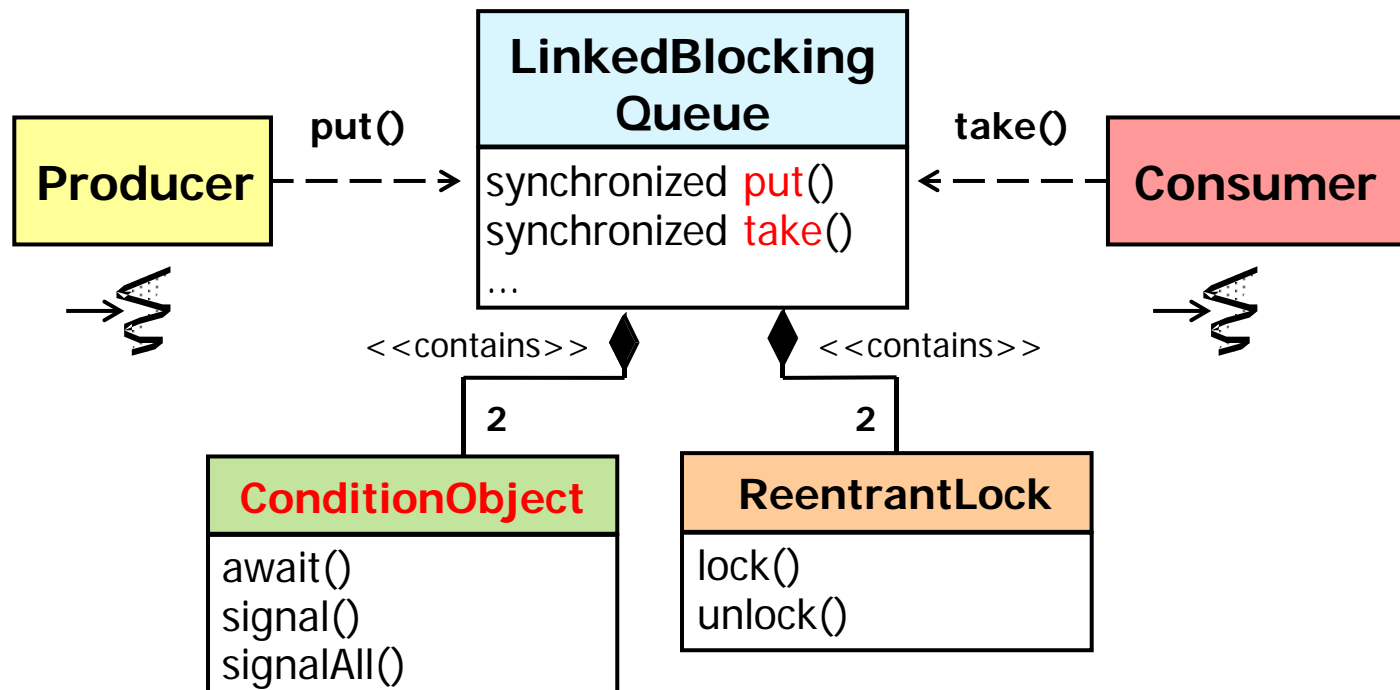


# Monitor Object

# POSA2 Concurrency

## Consequences

- + Concise programming model for concurrency control
- + Simplification of scheduling method execution
  - Synchronized methods use monitor conditions to determine when a thread should suspend or resume its execution & that of collaborating threads



# Monitor Object

# POSA2 Concurrency

**Consequences**

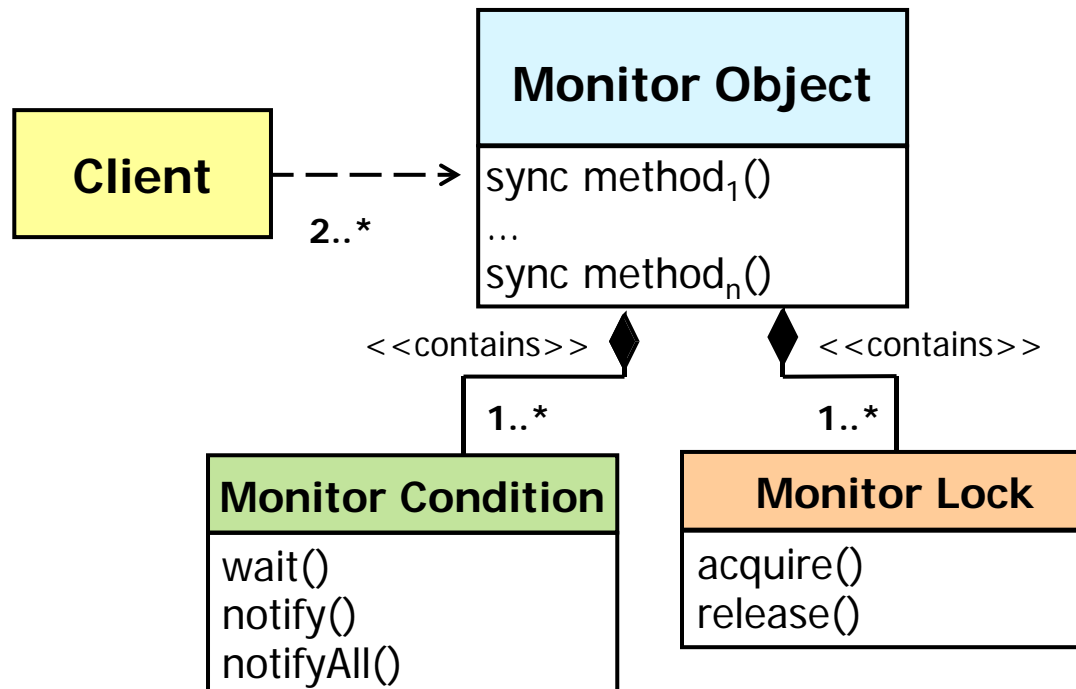


# Monitor Object

# POSA2 Concurrency

## Consequences

- Limited scalability
  - A single monitor lock can limit scalability due to increased contention when multiple threads serialize on a monitor object

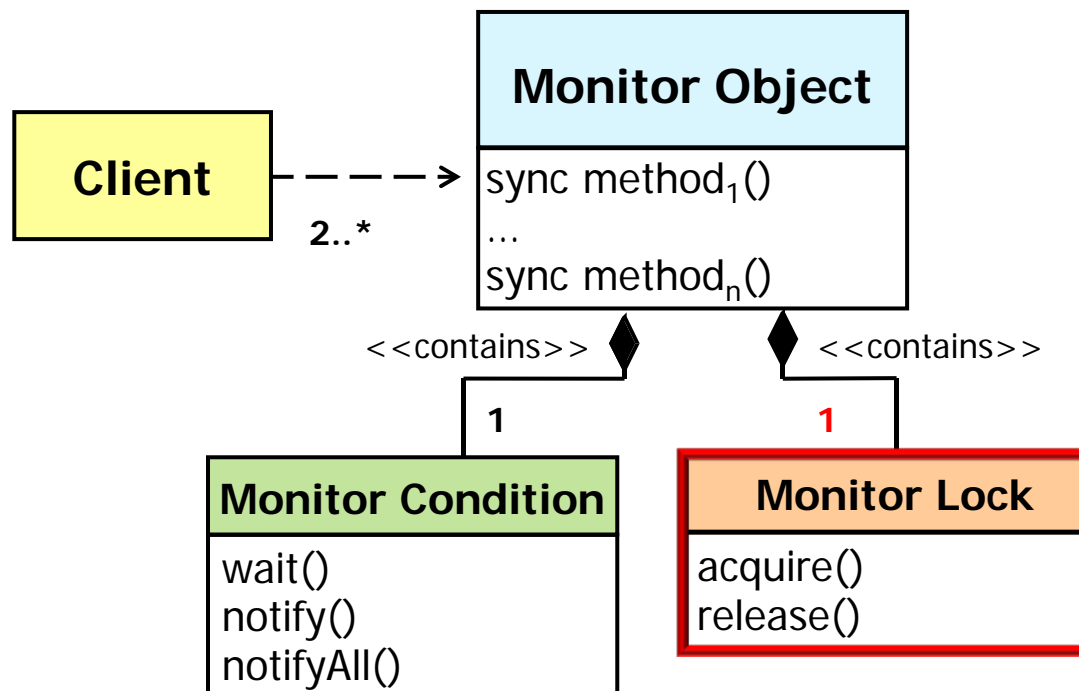


# Monitor Object

# POSA2 Concurrency

## Consequences

- Limited scalability
  - A single monitor lock can limit scalability due to increased contention when multiple threads serialize on a monitor object

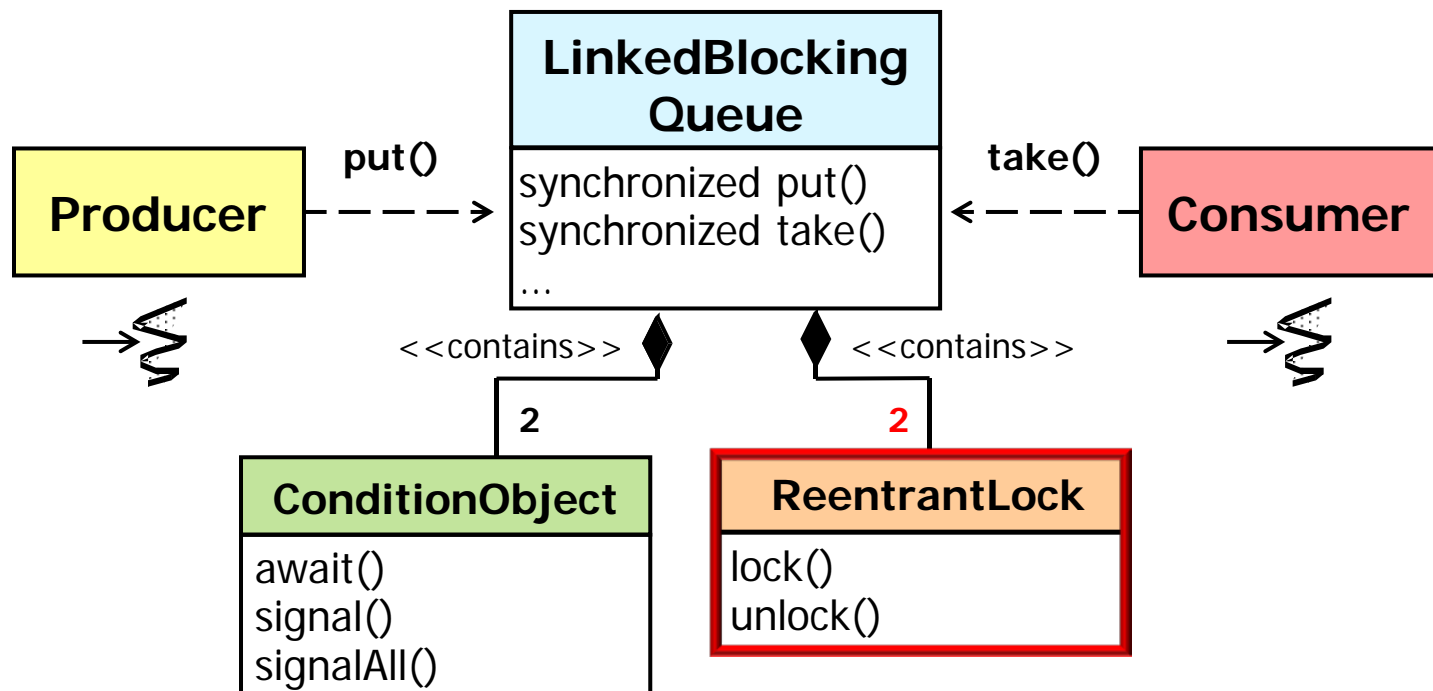


# Monitor Object

# POSA2 Concurrency

## Consequences

- Limited scalability
  - A single monitor lock can limit scalability due to increased contention when multiple threads serialize on a monitor object



See upcoming part 2 on "The Monitor Object pattern"



# Monitor Object

# POSA2 Concurrency

## Consequences

- Limited scalability
- Complicated extensibility semantics
  - Resulting from tight coupling between a monitor object's functionality & its concurrency control mechanisms

```
public class LinkedBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
        java.io.Serializable {
    public E take() ... {
        ...
        takeLock.lockInterruptibly();
        try {
            while (count.get() == 0) {
                notEmpty.await();
                x = dequeue();
                c = count.getAndDecrement();
                if (c > 1) notEmpty.signal();
            } finally { takeLock.unlock(); }
            if (c == capacity)
                signalNotFull();
            return x;
        }
        ...
    }
}
```

# Monitor Object

# POSA2 Concurrency

## Consequences

- Limited scalability
- Complicated extensibility semantics
  - Resulting from tight coupling between a monitor object's functionality & its concurrency control mechanisms

```
public class LinkedBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
        java.io.Serializable {
    public E take() ... {
        ...
        takeLock.lockInterruptibly();
        try {
            while (count.get() == 0) {
                notEmpty.await();
                x = dequeue();
                c = count.getAndDecrement();
                if (c > 1) notEmpty.signal();
            } finally { takeLock.unlock(); }
            if (c == capacity)
                signalNotFull();
            return x;
        }
        ...
    }
}
```

# Known Uses of the Monitor Object Pattern

# Monitor Object

# POSA2 Concurrency

## Known Uses

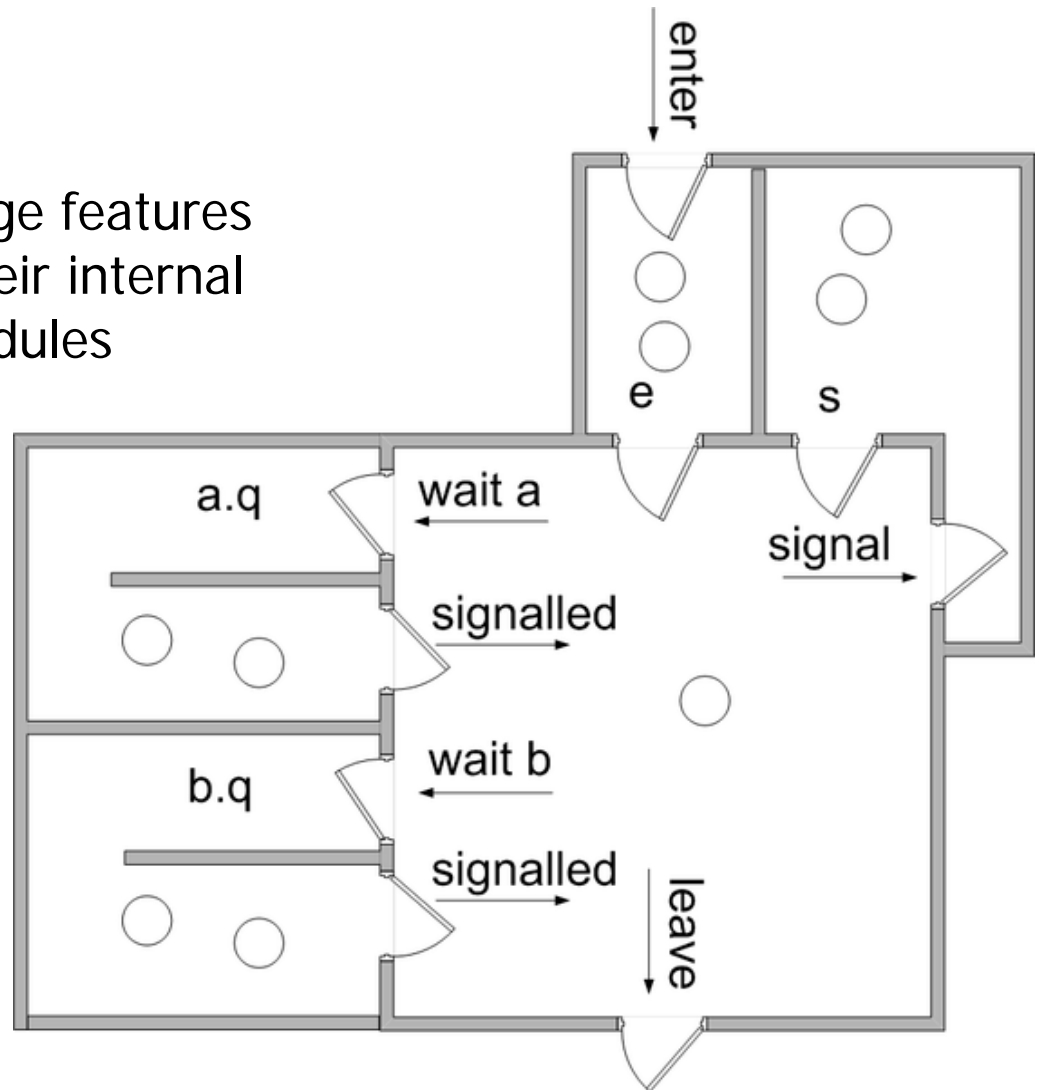


# Monitor Object

# POSA2 Concurrency

## Known Uses

- Dijkstra & Hoare-style Monitors
  - Defined programming language features to encapsulate functions & their internal variables into thread-safe modules

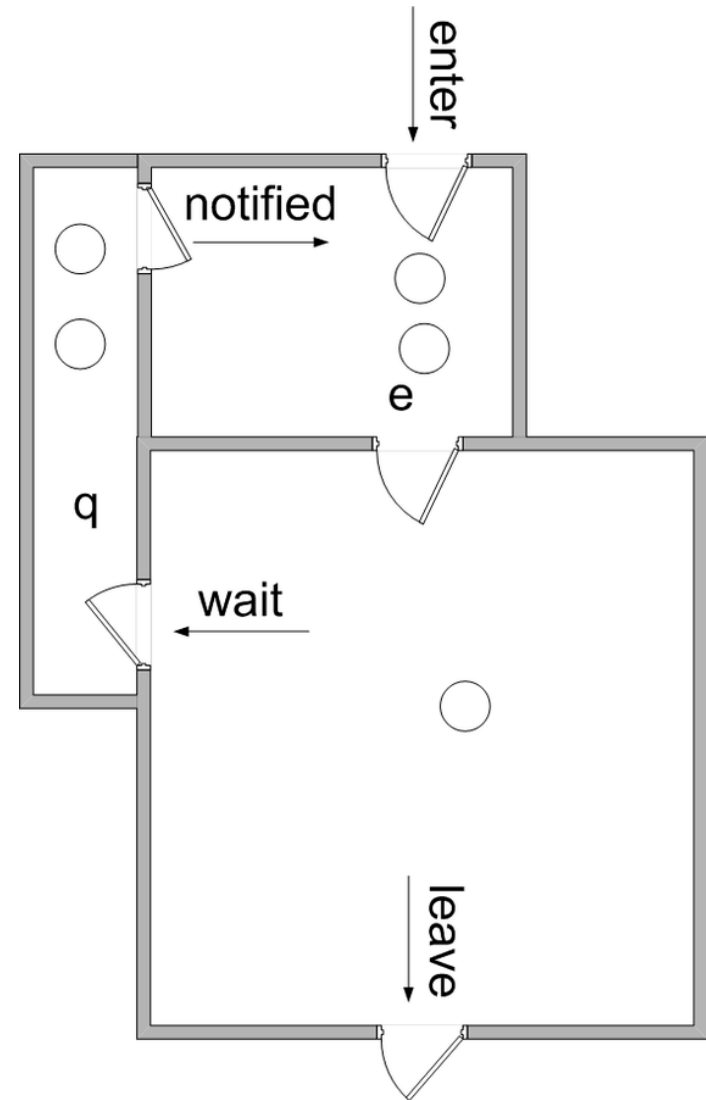


# Monitor Object

# POSA2 Concurrency

## Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks

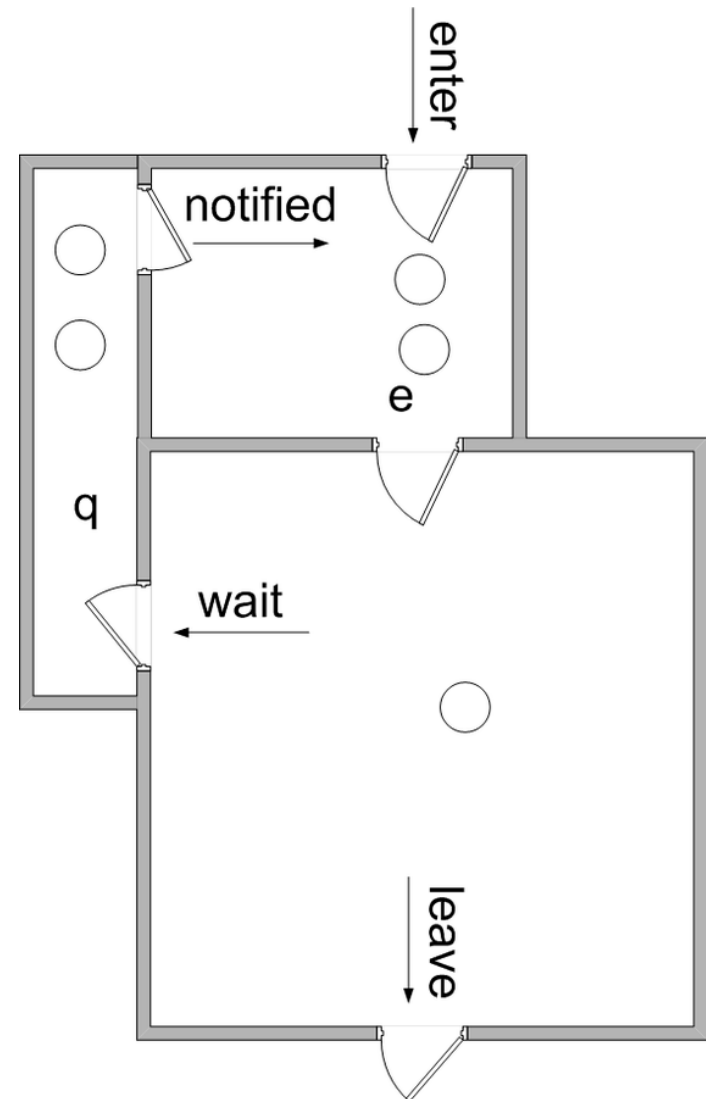


# Monitor Object

# POSA2 Concurrency

## Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
- Any Java object can be used as a monitor object



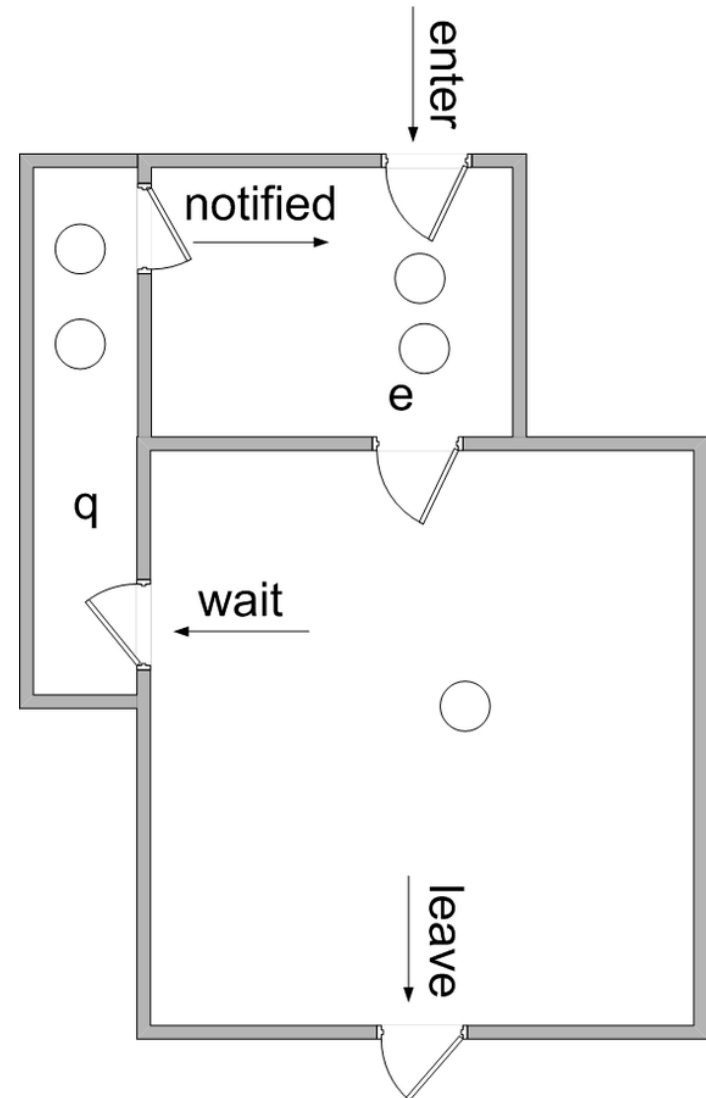
See earlier part on "Java Built-in Monitor Objects"

# Monitor Object

# POSA2 Concurrency

## Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
  - Any Java object can be used as a monitor object
- Java built-in monitor objects are convenient for simple use cases





# Monitor Object

# POSA2 Concurrency

## Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
  - Any Java object can be used as a monitor object
  - Java built-in monitor objects are convenient for simple use cases
- Although few synchronized methods/blocks are used in `java.util.concurrent`, the *Monitor Object* pattern is still widely applied

```
public class LinkedBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
        java.io.Serializable {
    public E take() ... {
        ...
        takeLock.lockInterruptibly();
        try {
            while (count.get() == 0) {
                notEmpty.await();
                x = dequeue();
                c = count.getAndDecrement();
                if (c > 1) notEmpty.signal();
            } finally { takeLock.unlock(); }
            if (c == capacity)
                signalNotFull();
            return x;
        }
        ...
    }
}
```

# Monitor Object

# POSA2 Concurrency

## Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
  - Any Java object can be used as a monitor object
  - Java built-in monitor objects are convenient for simple use cases
  - Although few synchronized methods/blocks are used in `java.util.concurrent`, the *Monitor Object* pattern is still widely applied

```
public class LinkedBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
        java.io.Serializable {
    public E take() ... {
        ...
        takeLock.lockInterruptibly();
        try {
            while (count.get() == 0) {
                notEmpty.await();
            }
            x = dequeue();
            c = count.getAndDecrement();
            if (c > 1) notEmpty.signal();
        } finally { takeLock.unlock(); }
        if (c == capacity)
            signalNotFull();
        return x;
        ...
    }
}
```

# Monitor Object

# POSA2 Concurrency

## Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
- C++ libraries provide building blocks for implementing monitor objects, e.g.
  - ACE

ACE Class
ACE_Guard ACE_Read_Guard ACE_Write_Guard
ACE_Thread_Mutex ACE_Process_Mutex ACE_Null_Mutex
ACE_RW_Thread_Mutex ACE_RW_Process_Mutex
ACE_Thread_Semaphore ACE_Process_Semaphore ACE_Null_Semaphore
ACE_Condition_Thread_Mutex ACE_Null_Condition

## Monitor Object

## POSA2 Concurrency

### Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
- C++ libraries provide building blocks for implementing monitor objects, e.g.
  - ACE
  - C++11

### C++11 threads, locks and condition variables

By Marius Bancila, 27 May 2013

★★★★★ 4.92 (34 votes)

Rate this: ☆☆☆☆☆



Prize winner in Competition "Best C++ article of May 2013"

### Threads

The `std::thread` class represents a thread of execution and is available in the `<thread>` header. `std::thread` can work with regular functions, lambdas and functors (a class implementing `operator()`). Moreover it allows you to pass any number of parameters to the thread function.

[Collapse](#) | [Copy Code](#)

```
#include <thread>

void func()
{
    // do some work
}

int main()
{
    std::thread t(func);
    t.join();

    return 0;
}
```

In this example `t` is a thread object representing the thread under which function `func()` runs. The call to `join` blocks the calling thread (in this case the main thread) until the joined thread finishes execution. If the thread function returns a value, it is ignored. However, the function can take any number of parameters.

[Collapse](#) | [Copy Code](#)

```
void func(int i, double d, const std::string& s)
{
    std::cout << i << ", " << d << ", " << s << std::endl;
}

int main()
{
    std::thread t(func, 1, 12.50, "sample");
    t.join();

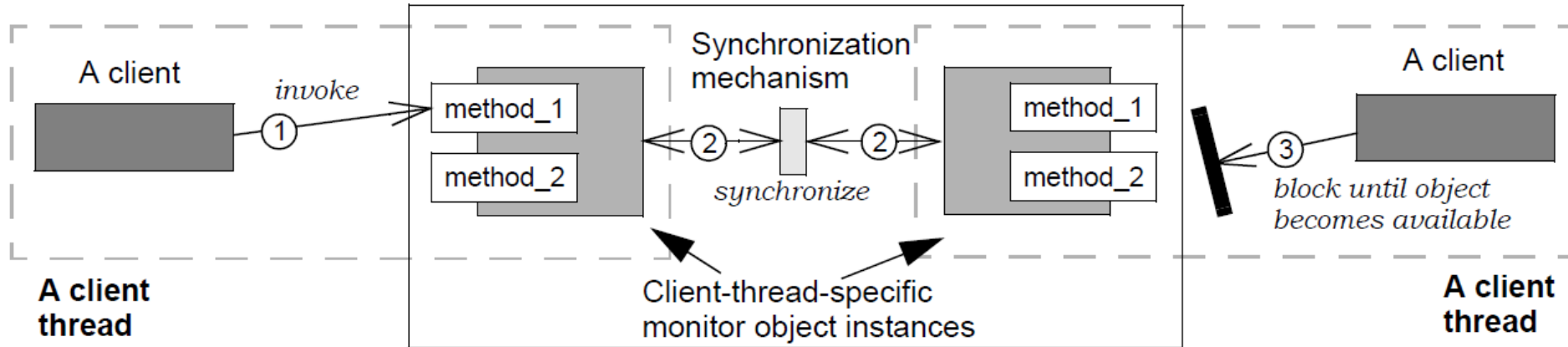
    return 0;
}
```

# Summary



# Summary

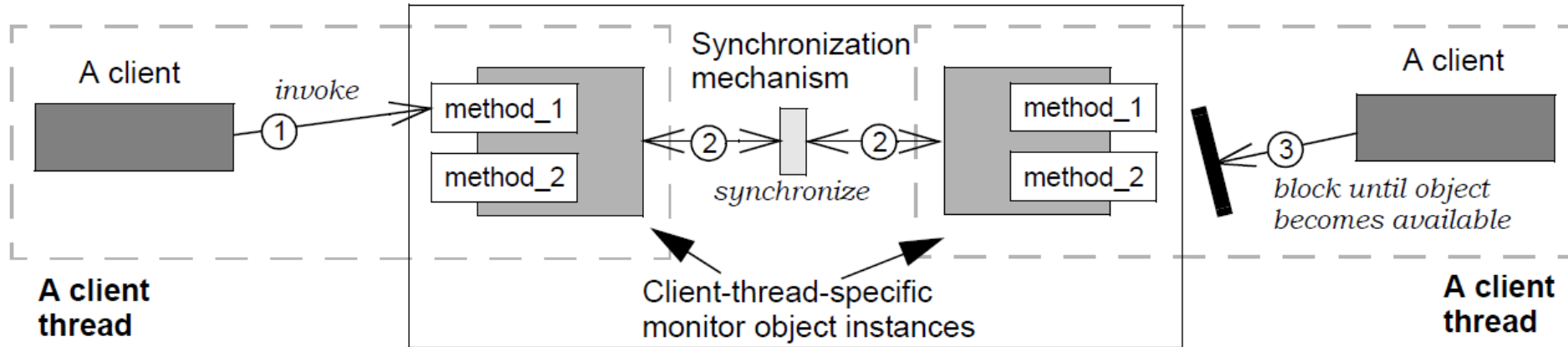
### Monitor object



- Concurrent software often contains objects whose methods are invoked by multiple client threads

# Summary

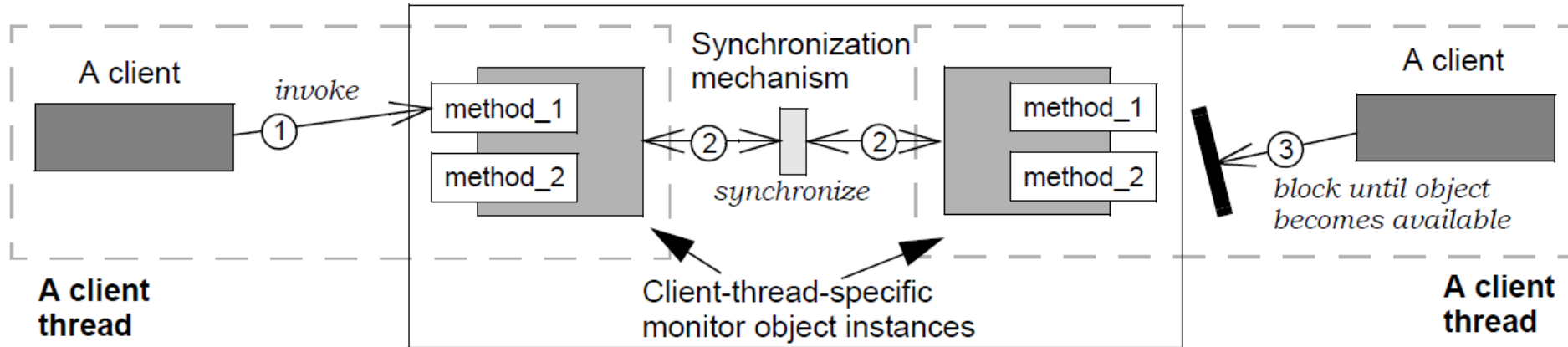
### Monitor object



- Concurrent software often contains objects whose methods are invoked by multiple client threads
- To protect the internal state of shared objects, it is necessary to synchronize & schedule access to them

# Summary

### Monitor object

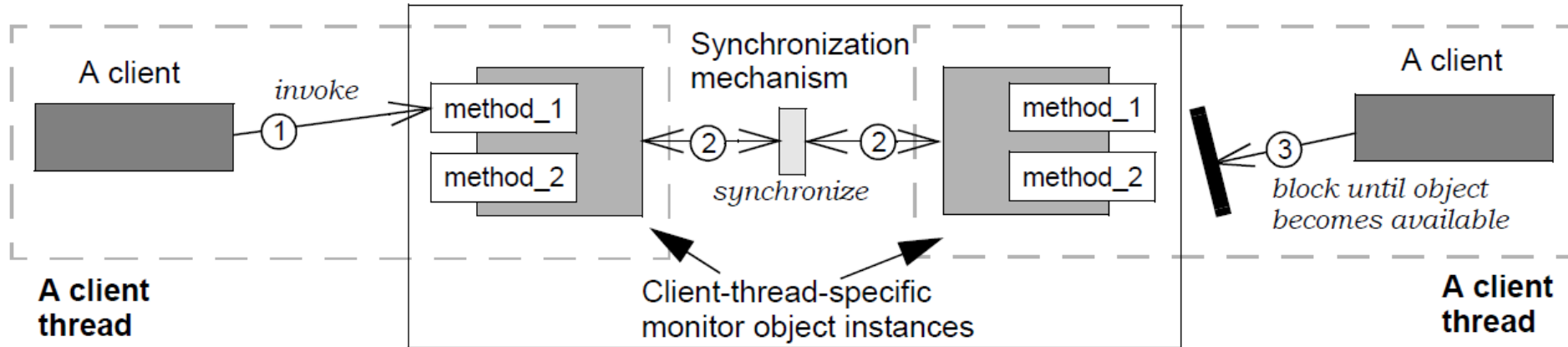


- Concurrent software often contains objects whose methods are invoked by multiple client threads
  - To protect the internal state of shared objects, it is necessary to synchronize & schedule client access to them
  - To simplify programming, however, clients should not need to distinguish between accessing shared & non-shared objects



# Summary

### Monitor object



- Concurrent software often contains objects whose methods are invoked by multiple client threads
- The *Monitor Object* pattern enables the sharing of object by client threads that cooperate to ensure a serialized—yet interleaved—execution sequence