Actor Object and Monitor Object

Parallelism

- allow processes to run concurrently
  - higher performance
  - less waiting/blocking
  - use of available hardware - more processors or cores
- problems
  - resource sharing between processes (race conditions, deadlocks, ...)
  - non-determinism

Active object

- concurrency pattern
- decouples method execution from method invocation
- simplify synchronized access to objects with own threads of control

Problem - logger

```
int main() {
    Logger logger = Logger();

    for (int i = 0; i < 1000000; i++) {
        logger.log(LogLevel::LOG, "Sqrt of number" +  std::to_string(i) + " is "
+                       std::to_string(std::sqrt(i)));
    }

    logger.close();

    return 0;
}
```

Threaded logger

```
class ThreadLogger {
private:
    Logger* internalLogger;
    std::mutex lock;

public:
    ThreadLogger() {
        internalLogger = new Logger();
    }

    void log(LogLevel level, std::string message) {
        std::thread([this, level, message]() {
            lock.lock();
            internalLogger->log(level, message);
            lock.unlock();
        }).detach();
    }

    void close() {
        internalLogger->close();
    }
};
```

Add scheduler

```
class LoggerScheduler {
private:
    std::queue<Message> queue;
    std::mutex lock;
    Logger servant;


public:
    void insert(Message mr) {
        lock.lock();
        queue.push(mr);
        lock.unlock();
    }

    virtual void dispatch();
};
```

Making logs into requests

```
class LoggerMethodRequest {
public:
    int priority = -1;
    int ordNum = -1;
    virtual void execute(Logger logger) = 0;
};
```

```
class LoggerLogRequest : LoggerMethodRequest {
 public:
    int priority = 3;
    int ordNum = 0;
    LogLevel level;
    std::string message;
    LoggerLogRequest(LogLevel level, std::string
                                                  message) {
        ordNum = ordNumCounter++;
        this->level = level;
        this->message = message;
    }

    void execute(Logger logger) {
        logger.log(level, message);
    }
};
```

Adding proxy

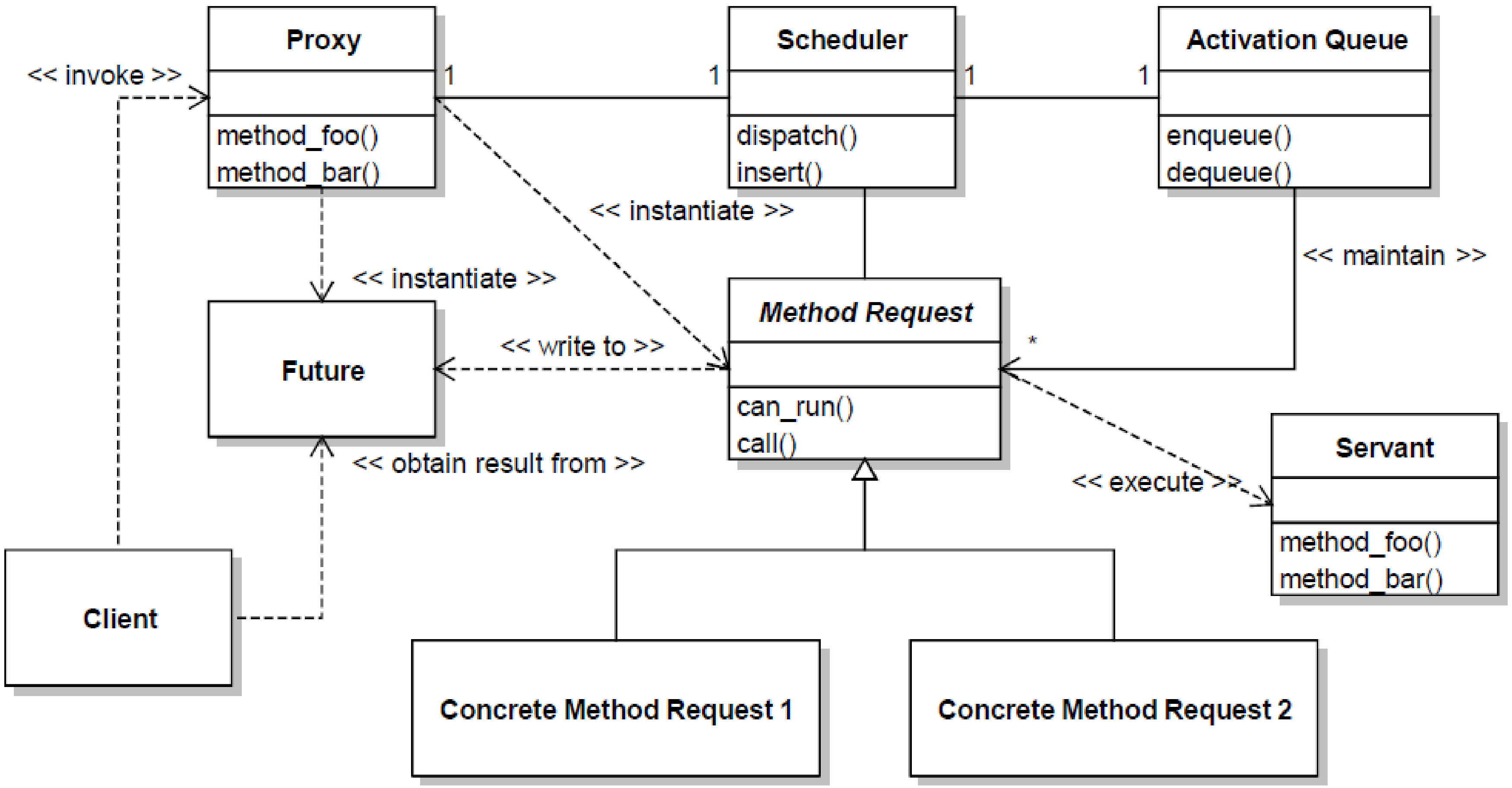
```
class Proxy {
public:
    void log(const std::string msg) {
        LoggerMethodRequest *mr = new LoggerLogRequest(LogLevel::LOG, msg);
        scheduler_.insert(mr);
    }

    Message_Future get() {
        Message_Future result;
        LoggerMethodRequest *mr = new Get(servant_, result);
        scheduler_.insert(mr);
        return result;
    }
private:
    Logger_Servant servant_;
    LoggerScheduler scheduler_;
};
```

Formal structure

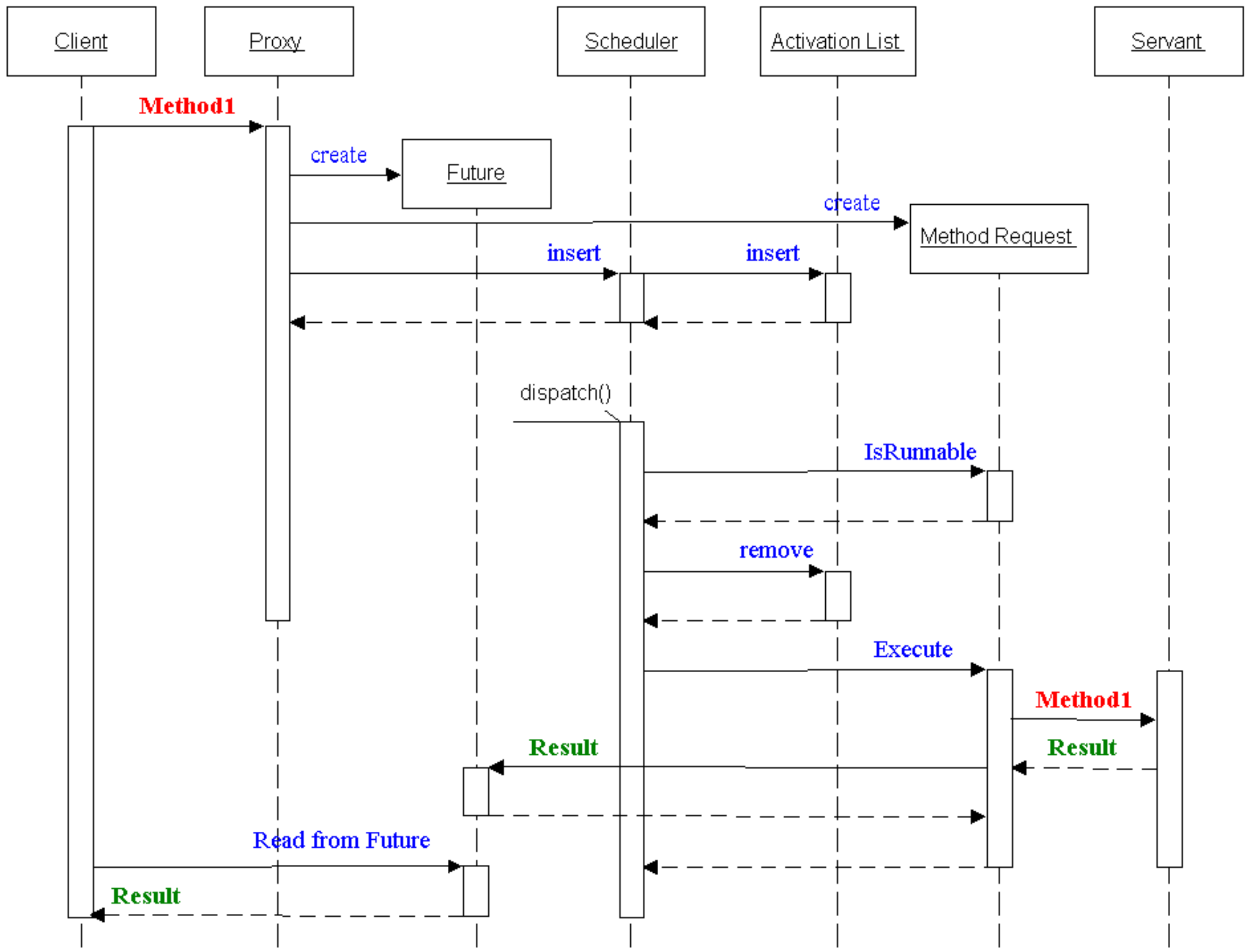
- **proxy**
  - creates method request and provides future
- **method request**
  - defines context for the method
- **activation queue**
- **scheduler**
  - decides which request to execute next from the queue
- **servant**
  - executes the implementation of the request
- **future**
  - contains the result of the request

Overview



Proxy
method_foo()
method_bar()
Scheduler
dispatch()
insert()
Activation Queue
enqueue()
dequeue()
<< invoke >>
1
1
1
1
<< instantiate >>
<< instantiate >>
<< maintain >>
Future
Method Request
can_run()
call()
*
<< write to >>
<< obtain result from >>
<< execute >>
Servant
method_foo()
method_bar()
Client
Concrete Method Request 1
Concrete Method Request 2

The flow



Client
Proxy
Scheduler
Activation List
Servant
Method1
create
Future
create
Method Request
insert
insert
dispatch()
IsRunnable
remove
Execute
Method1
Result
Result
Read from Future
Result

Example - restaurant

- **proxy** as the waiter, who continuously takes orders and delivers them to the kitchen
- worklist in the kitchen as **activation list**
- **scheduler** is the chef, who decides how to make the orders most efficiently
- **servant** is the cook

Active object

- pros
  - handles synchronization and hides it from the client (apart from futures)
  - supports complex scheduling heuristics
  - easily scalable
- cons
  - fair amount of added complexity
  - complicated debugging

Connections with other patterns

- **proxy**
- method request as **command** instance
- scheduler as **command processor** instance
- future can be implemented as a **counted pointer**

Monitor object

- concurrency pattern
- synchronizes concurrent method execution so that only one method at a time runs within an object
- simplify synchronized access to objects without own threads of control

Example - counter

```cpp
// Monitor Object
class Counter {
private:
    int count_;
    std::mutex monitor_lock_;
    // std::condition_variable monitor_condition_;
public:
    Counter() {
        count_ = 0;
    }

    void increment() {
        std::lock_guard<std::mutex> lock(monitor_lock_);
        count_++;
    }

    int getCount() {
        std::lock_guard<std::mutex> lock(monitor_lock_);
        return count_;
    }
};
```

```cpp
// Client code
class Client {
    void main() {
        auto counter = new Counter();
        // Create and start multiple
           threads that increment
           the counter
    }
};
```

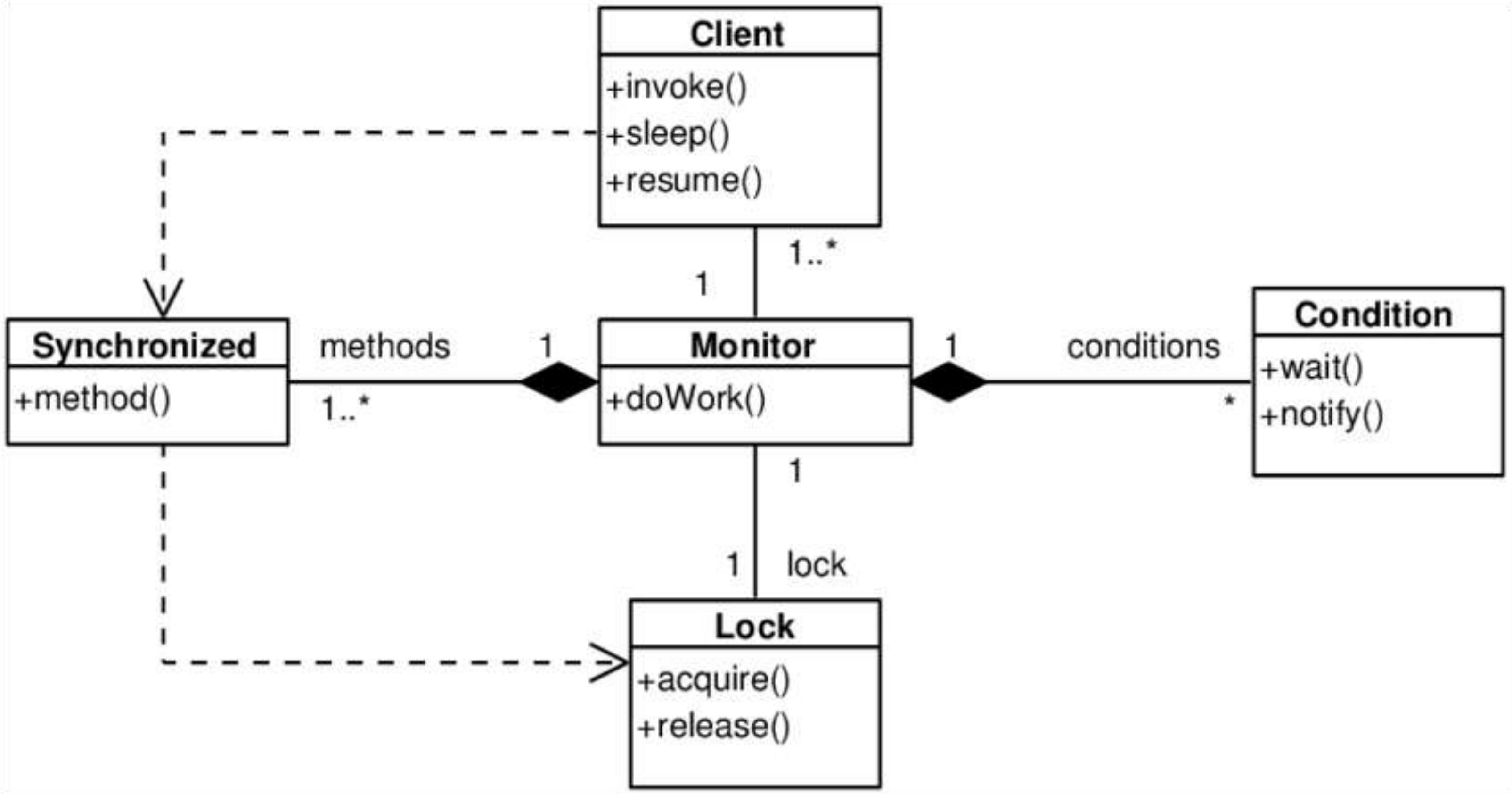
Formal structure

- **synchronized method**
  - publicly accessible
- **monitor lock**
  - ensures only one method of monitor is executed at the same time
- **monitor condition**
  - determines which synchronized methods should be suspended and reactivated

Conditions

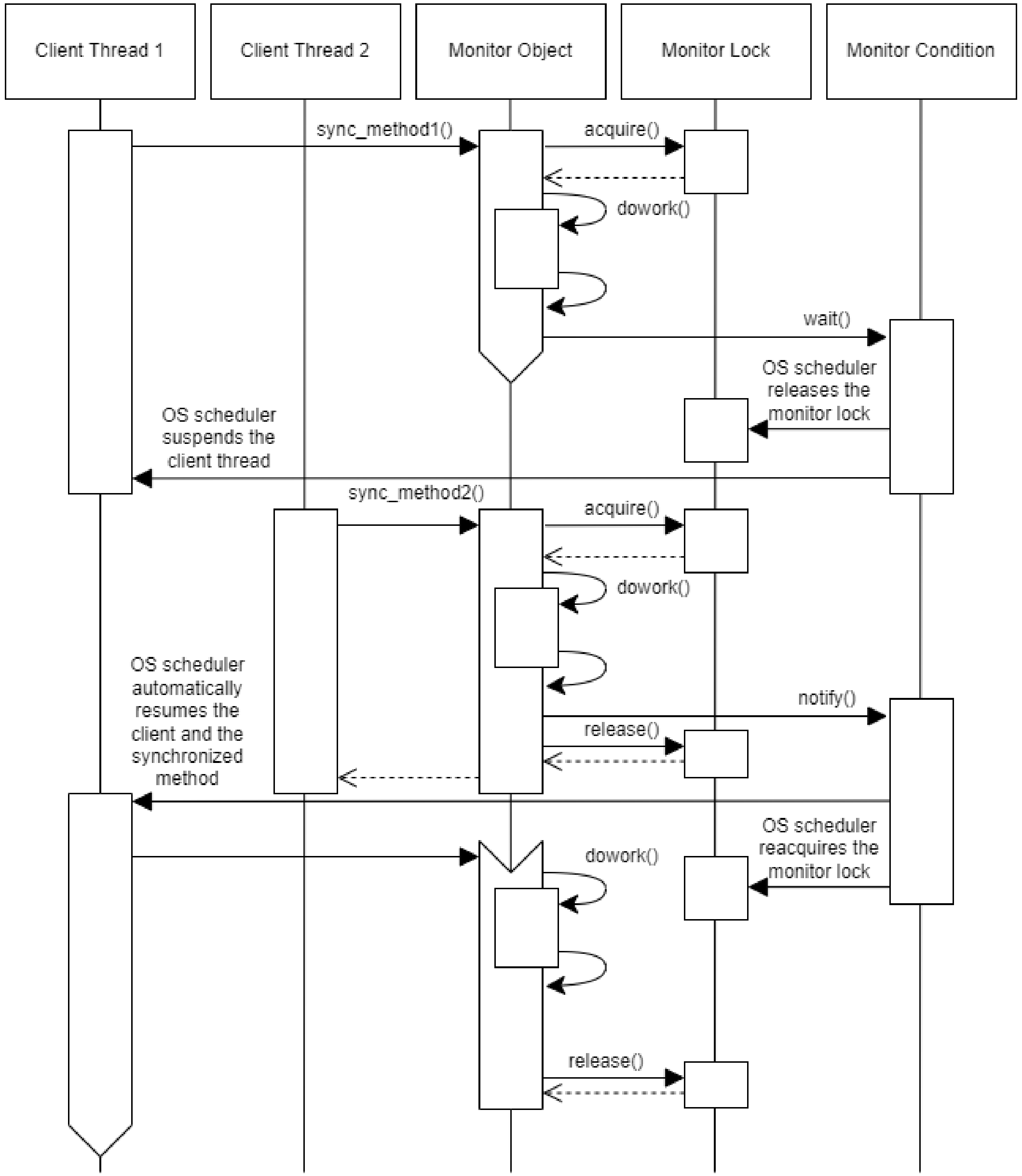
- implemented with condition variables
  - Java has java.util.concurrent.locks.Condition
  - C# has System.Threading.Monitor
  - C++ has std::condition_variable

Overview



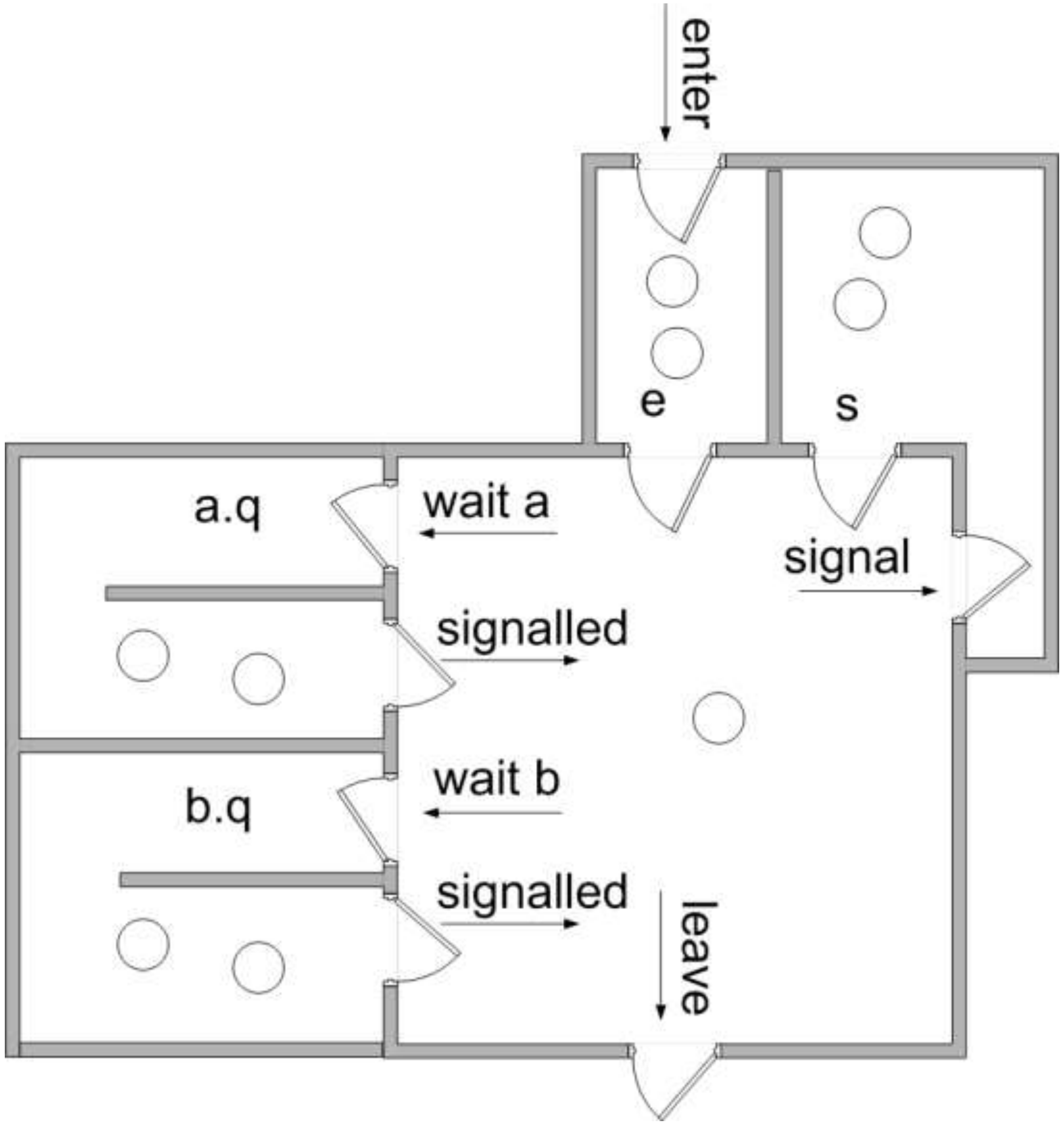
Client
+invoke()
+sleep()
+resume()
1..*
1
Synchronized
+method()
methods
1..*
1
Monitor
+doWork()
1
conditions
*
Condition
+wait()
+notify()
1
1
lock
Lock
+acquire()
+release()

The flow



Client Thread 1
Client Thread 2
Monitor Object
Monitor Lock
Monitor Condition
sync_method1()
acquire()
dowork()
wait()
OS scheduler releases the monitor lock
OS scheduler suspends the client thread
sync_method2()
acquire()
dowork()
notify()
release()
OS scheduler automatically resumes the client and the synchronized method
OS scheduler reacquires the monitor lock
dowork()
release()

Another example – fast-food restaurant



Monitor object

- pros
  - handles synchronization and hides it from the client
  - lighter
- cons
  - synchronization tightly coupled with object funcionality
  - limits scalability
  - lacks own thread of control and, therefore has no control over the order synchronized methods access it

Active vs. Monitor object

Active

- complex
- runs on own thread
- supports more complex schedulling
- asynchronous retrieval of results

Monitor

- lighter
- runs on clients thread
- usually used for smaller objects