Inter-thread Communication and Signalling

David Flynn
SOFTWARE ENGINEER, FLYNN IT LTD

The Need for Signalling

No work for thread to do just yet

Need to supply thread with additional data during an operation

ACKs

Wait and Mutex (Monitor) Release

Implement producer/ consumer model

Demonstrate BlockingQueue

Producer/Consumer Model

Messages sent from one or more producers to one or more consumers

- Consumer uses the message
- Producer sends it

Consumers carry out the work on behalf of the producers

Separation

Allows to offload work onto flexible number of workers

Separating concerns – giving higher cohesion

- Simplifies APIs
- Makes code easier to write, debug and maintain

Decoupling – gives looser coupling

- Can change one without affecting other
- Helps when writing unit tests

Thread Pools

Arrange threads in groups or pools

To carry out a particular task

For now we'll use an array of threads

Next module shows how to use ThreadPoolExecutors

Bar-Restaurant Requirements

Each thread in the customer pool represents a party of diners

- Customers arrive randomly and wait to be seated by waiters
- May only be served if they don't arrive too late
- If they arrive by closing time they may eat

Any waiter [in the waiter pool] can carry out any request

Bar-Restaurant Requirements

Customers request waiters:

- To order food
- Request to bring meal
- To bring the cheque

Customer pays and leaves

Bar-Restaurant Requirements

At closing time:

- Unseated customers are turned away
- Waiters can go home
- Bar closes

Spinlock Code Snippet

```
public static volatile boolean condition = false;
...
while (!condition) {} // Wait until another thread sets
```

wait()/ notify()/ notifyAll()

Provides basic signalling

Threads that want a signal call wait

- Sleep until another thread calls notify()/ notifyAll()

notify() wakes one thread waiting on the object

notifyAll() wakes all threads waiting on the object

Wait Set

Waiting requires a condition variable aka wait set

- This queues the waiting threads
- Objects are associated with wait sets

Use synchronized(object) before calling wait()/notify()/notifyAll()

- Otherwise throws IllegalMonitorStateException

Wait Set Miscellanea

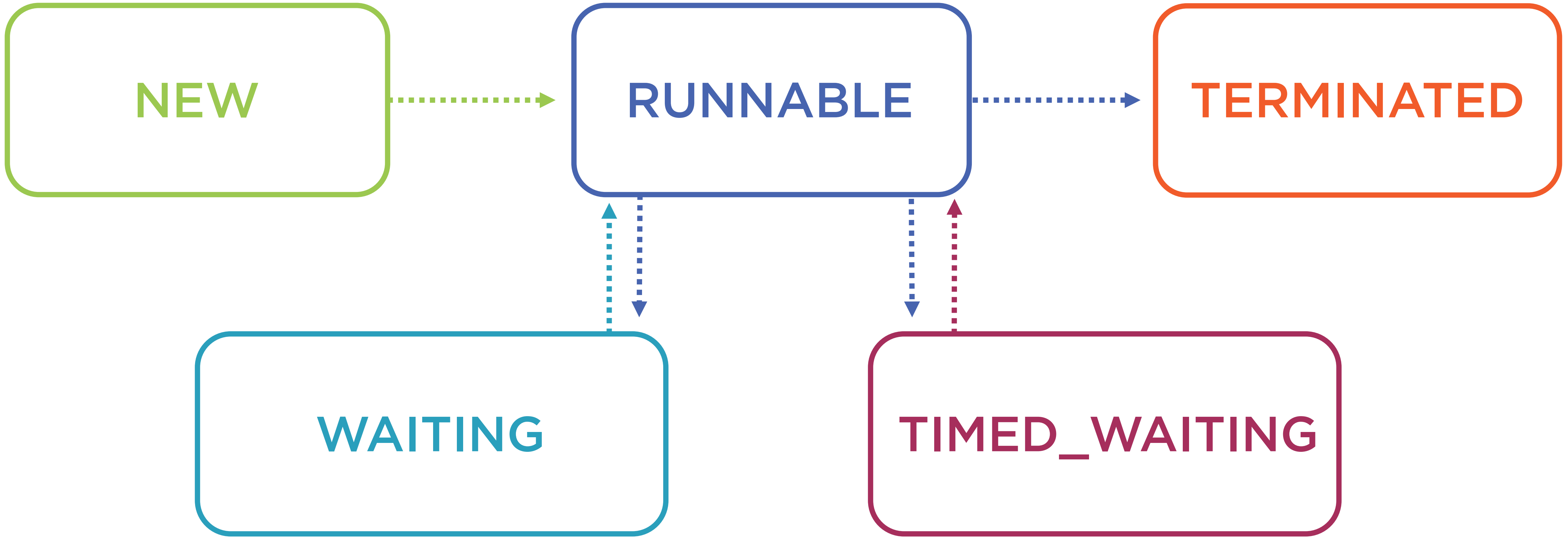
Shouldn't use wait on Thread or anything which extends it

Wait has timeout versions [ms, ms & ns] like join

- Timeout of 0 calls non-timeout version
- But not in TimeUnit version which doesn't wait
- Caller to work out if timed out or interrupted

Wait throws InterruptedException

The Thread State Machine

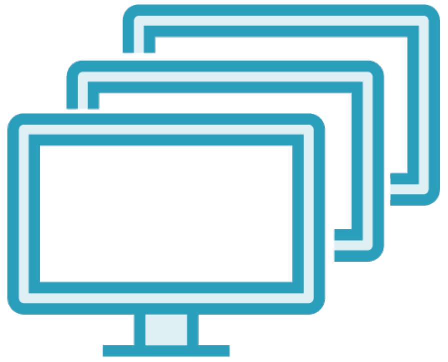


Dangers to Watch For

IllegalMonitorStateException is thrown if:

- Forgetting to synchronize
- Synchronizing on the wrong object

Waiting and Monitors



When thread added to wait set
- Releases its monitor
- Other threads can call wait or notify

When thread is awakened from wait
- Awakened thread reacquires monitor

Waiting and Monitors

Notifying thread should release monitor as soon as it can

- Otherwise may cause starvation

notifyAll() causes all waiting threads to reacquire monitor

- Exit from wait() - single threaded
- Wait shouldn't hold it for too long

Beware of race conditions due to releasing the monitor

Spurious Wakeups

Due to hardware optimizations

Causes threads to wake from wait() without notify() being called

- Which might cause subtle bugs

Notification Without Waiting Threads

Notifications are not stored

- Signal is lost
- Can lead to threads waiting forever

Fix by enclosing wait() in a while loop

Lost Notification Problem

Interruption at same time as notification

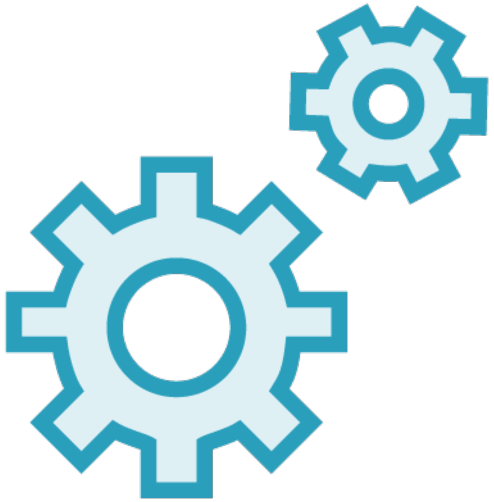
- Can't tell if notified [but can look at the queue]

No issue when interruption means don't process any more work

If need to handle work

- Remember was interrupted while doing outstanding work

notify() vs notifyAll()



Waking mechanism may not be fair

- On notify, same thread could awaken causing starvation

If waiting for different conditions on the same object

- Wrong thread could awaken and signal lost
- Must use notifyAll() here

BlockingQueue Introduction

BlockingQueue is an interface

- LinkedBlockingQueue – linked list version which doesn't get full
- ArrayBlockingQueue – array list version of fixed size

Fixed size can prevent queue growing too large

- When producers leaving more work than consumers can consume

BlockingQueue Methods

	Insertion	Removal	Inspect Head
Returns null	offer(e)	poll()	peek()
Throws Exception	add(e)	remove()	element()
Blocking	put(e)	take()	N/A
Blocking and Timeout	offer(e, time, unit)	poll(time, unit)	N/A

BlockingQueue Methods

int drainTo (collection [, max_elements])

Producer/Consumer Model

Producers send messages to consumers to consume

Splitting into producer/consumer:

- Separates concerns – higher cohesion
- Interacting via a queue – loose coupling

The Four Implementations

Constantly polling, spinlocks consuming CPU

Sleeping between polling, wait too often or decreased responsiveness

wait()/ notify()

BlockingQueue

wait()/ notify()/ notifyAll()

wait()/ notify()/ notifyAll() are methods on Object

- Have to synchronize on the object they are called on

notify() signals one thread waiting on the object

notifyAll() signals all threads waiting on the object

Use of While Loop Condition

Handling spurious wake-ups and notification signal race condition

- Signalling thread updates condition while synchronized

Handling interrupts and timeouts

- Danger of interrupt and signal arriving at the same time

wait()/ notify() Exposes Us to Other Problems

Forgetting to synchronize on the object

Synchronizing on the wrong object

- IllegalMonitorStateException - only discovered at runtime

Missing notifications

Use notify() when threads waiting on different conditions

- Lead to 'deadlock'; no further signal to wake threads up

BlockingQueue

Makes life so much easier

Should use higher level implementations where possible

- Low level wait()/notify() can be hard to get right