Threads

Weng Kai

Explicit Lock

- synchronized是内部锁，由JVM执行
- java.util.concurrent.locks.Lock接口定义了显式锁

void	**lock**() Acquires the lock.
void	**lockInterruptibly**() Acquires the lock unless the current thread is **interrupted**.
Condition	**newCondition**() Returns a new **Condition** instance that is bound to this Lock instance.
boolean	**tryLock**() Acquires the lock only if it is free at the time of invocation.
boolean	**tryLock**(long time, **TimeUnit** unit) Acquires the lock if it is free within the given waiting time and the current thread has not been **interrupted**.
void	**unlock**() Releases the lock.

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html

Basic Snippet

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this
lock
} finally {
    l.unlock();
}
```

ReentrantLock

- A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock.

```
class X {
   private final ReentrantLock lock = new ReentrantLock();
   // ...

   public void m() {
     lock.lock();  // block until condition holds
     try {
       // ... method body
     } finally {
       lock.unlock()
     }
   }
 }
```

Fair or Not

- The constructor for this class accepts an optional fairness parameter. When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order. Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. Note however, that fairness of locks does not guarantee fairness of thread scheduling. Thus, one of many threads using a fair lock may obtain it multiple times in succession while other active threads are not progressing and not currently holding the lock. Also note that the untimed tryLock method does not honor the fairness setting. It will succeed if the lock is available even if other threads are waiting.

内部锁vs显式锁

- 内部锁基于代码块，无法跨越函数
- 内部锁没有公平性唤醒
- 内部锁对多重锁的支持不好
- 显式锁有tryLock和带超时的锁
- 显式锁支持读-写分离的锁

ReadWriteLock

- A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive.

`Lock`	`readLock()` Returns the lock used for reading.
`Lock`	`writeLock()` Returns the lock used for writing.

ReadWriteLock

	获得条件	排他性	作用
readLock.lock()	没有被写锁	允许其他线程读 不允许其他线程写	多个线程可以同时读取共享，同时禁止有人读的时候写入
writeLock.lock()	没有被写锁&&没有被读锁	禁止一切其他访问	线程独占方式写

- A read-write lock allows for a greater level of concurrency in accessing shared data than that permitted by a mutual exclusion lock. It exploits the fact that while only a single thread at a time (a writer thread) can modify the shared data, in many cases any number of threads can concurrently read the data (hence reader threads). In theory, the increase in concurrency permitted by the use of a read-write lock will lead to performance improvements over the use of a mutual exclusion lock. In practice this increase in concurrency will only be fully realized on a multi-processor, and then only if the access patterns for the shared data are suitable.

- Whether or not a read-write lock will improve performance over the use of a mutual exclusion lock depends on the frequency that the data is read compared to being modified, the duration of the read and write operations, and the contention for the data - that is, the number of threads that will try to read or write the data at the same time. For example, a collection that is initially populated with data and thereafter infrequently modified, while being frequently searched (such as a directory of some kind) is an ideal candidate for the use of a read-write lock. However, if updates become frequent then the data spends most of its time being exclusively locked and there is little, if any increase in concurrency. Further, if the read operations are too short the overhead of the read-write lock implementation (which is inherently more complex than a mutual exclusion lock) can dominate the execution cost, particularly as many read-write lock implementations still serialize all threads through a small section of code. Ultimately, only profiling and measurement will establish whether the use of a read-write lock is suitable for your application.

ReentrantReadWriteLock

- Acquisition order
    - This class does not impose a reader or writer preference ordering for lock access. However, it does support an optional fairness policy (approximately arrival-order policy).
        - A thread that tries to acquire a fair read lock (non-reentrantly) will block if either the write lock is held, or there is a waiting writer thread
- Lock downgrading
    - Reentrancy also allows downgrading from the write lock to a read lock, by acquiring the write lock, then the read lock and then releasing the write lock. However, upgrading from a read lock to the write lock is not possible.

Downgrade

```
class CachedData {
   Object data;
   volatile boolean cacheValid;
   final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

   void processCachedData() {
     rwl.readLock().lock();
     if (!cacheValid) {
        // Must release read lock before acquiring write lock
        rwl.readLock().unlock();
        rwl.writeLock().lock();
        try {
          // Recheck state because another thread might have
          // acquired write lock and changed state before we did.
          if (!cacheValid) {
            data = ...
            cacheValid = true;
          }
          // Downgrade by acquiring read lock before releasing write lock
          rwl.readLock().lock();
        } finally {
          rwl.writeLock().unlock(); // Unlock write, still hold read
        }
     }

     try {
       use(data);
     } finally {
       rwl.readLock().unlock();
     }
   }
}
```

```
class RWDictionary {
    private final Map<String, Data> m = new TreeMap<String, Data>();
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    public Data get(String key) {
        r.lock();
        try { return m.get(key); }
        finally { r.unlock(); }
    }
    public String[] allKeys() {
        r.lock();
        try { return m.keySet().toArray(); }
        finally { r.unlock(); }
    }
    public Data put(String key, Data value) {
        w.lock();
        try { return m.put(key, value); }
        finally { w.unlock(); }
    }
    public void clear() {
        w.lock();
        try { m.clear(); }
        finally { w.unlock(); }
    }
}
```

volatile

- The Java volatile keyword is used to mark a Java variable as "being stored in main memory”.

```
public class SharedObject {
    public volatile int counter = 0;
}
```

- all writes to the counter variable will be written back to main memory immediately. Also, all reads of the counter variable will be read directly from main memory

Full volatile Visibility Guarantee

- If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B after it has read the volatile variable.

- If Thread A reads a volatile variable, then all all variables visible to Thread A when reading the volatile variable will also be re-read from main memory.

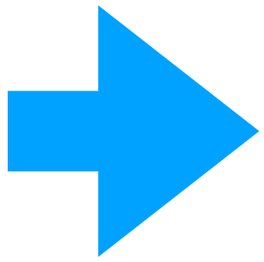
- The full volatile visibility guarantee means, that when a value is written to days, then all variables visible to the thread are also written to main memory. That means, that when a value is written to days, the values of years and months are also written to main memory.

```
public class MyClass {
    private int years;
    private int months
    private volatile int days;


    public void update(int years, int months, int days){
        this.years  = years;
        this.months = months;
        this.days   = days;
    }
}
```

Instruction Reordering

- The Java VM and the CPU are allowed to reorder instructions in the program for performance reasons, as long as the semantic meaning of the instructions remain the same

```
int a = 1;
int b = 2;

a++;
b++;
```



```
int a = 1;
a++;

int b = 2;
b++;
```

Happens-Before Guarantee

- Reads from and writes to other variables cannot be reordered to occur after a write to a volatile variable, if the reads / writes originally occurred before the write to the volatile variable. The reads / writes before a write to a volatile variable are guaranteed to "happen before" the write to the volatile variable.

- Reads from and writes to other variables cannot be reordered to occur before a read of a volatile variable, if the reads / writes originally occurred after the read of the volatile variable.

Atomic Operation

- Any operation over double and long is not an atomic one, unless the variable is declared as volatile

volatile

- visibility
- happen-before guaranteed
- Atomic operation for double and long

Volatile Object?

- volatile Student s ...
  - s the pointer is volatile, not the whole object
- All members are volatile?
  - Successive atomic operations are not atomic

Singleton

- Make the constructor private, and
- Provide a static method to retrieve that only object
  - Instantiated at loading, or
  - Instantiated at first retrieving

Issue

- if (theObject == null) theObject = new Server();
    - Not an atomic operation
- It’s too heavy to add lock at this retrieve method
    - It’s a rare situation

Two Stage Test

- If the object has been born, there's no need to generate a new one.
- Otherwise use a sort of lock to protect it.
- Use volatile to avoid instruction re-order

Other Ways

- static instantiation

- Use enum

Atomic Types

- CAS: compare and swap
  - If-then-act
- java.util.concurrent.atomic
  - read-modify-write

Atomics

primitives	AtomicInteger, AtomicLong, AtomicBoolean
arrays	AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray

Methods of AtomicLong

- get()
- getAndIncrement()
- getAndDecrement()
- incrementAndGet()
- decrementAndGet()
- set()

wait/notify

- Object.wait() puts the running thread into WAIT state
- Object.notify() wakes up one of the waiting threads over this object
- Object.notifyAll() wakes up all the waiting threads over this object
- Object.wait(long ms) will hold the thread for at least ms

Producer and consumer

- is a pattern that one thread produces data and the other one read it. There must be a shared variable for transportation and a flag to indicate the data is valid or has been read.

Case Study: FlagComm.java

Wait() & notify()

- wait() and notify() of Object
  Every object can have a thread pool. A thread can call wait() to join the pool and call notify() to leave the pool.
  Case Study: WaitComm.java

Wait/notify

- 等待线程对保护条件的判断、wait()的调用总是应该放在相应对象所引导的临界区中的一个循环语句之中

- 等待线程对保护条件的判断、 Object. wait()的执行以及目标动作的执行必须放在同一个对象（内部锁）所引导的临界区之中

- Object. wait()暂停当前线程时释放的锁只是与该 wait方法所属对象的内部锁。当前线程所持有的其他内部锁、显式锁并不会因此而被释放。

Issues

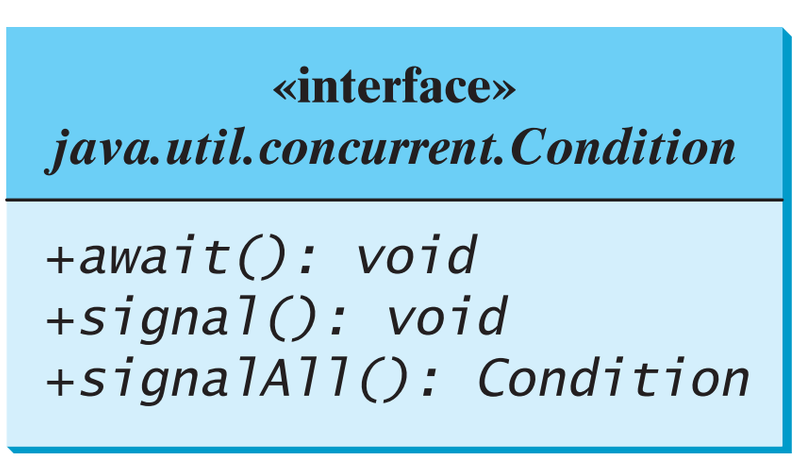
- Wake up too soon
- Missed signal
- Spurious wakeup

Condition

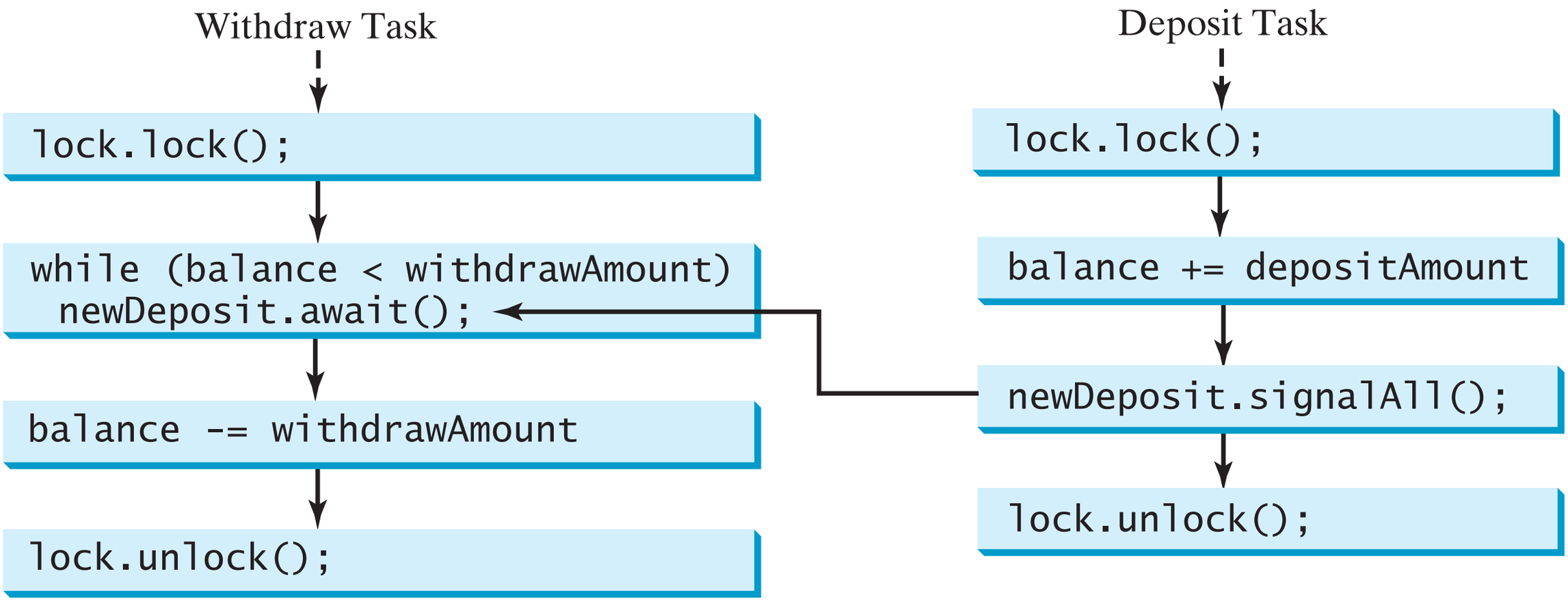
- java.util.concurrent.locks.Condition
  - await —> wait
  - signal —> notify
  - signalAll —> notifyAll
- Condition Lock.newCondition()

Condition

- Conditions are objects created by invoking the newCondition() method on a Lock object.



ThreadCooperation.java


Withdraw Task
lock.lock();
while (balance < withdrawAmount)
newDeposit.await();
balance -= withdrawAmount
lock.unlock();
Deposit Task
lock.lock();
balance += depositAmount
newDeposit.signalAll();
lock.unlock();

- 调用任意一个显式锁实例的 newCondition方法可以创建一个相应的Condition接口。Condition. await()/ signal()也要求其执行线程持有创建该 Condition实例的显式锁

- Condition实例也被称为条件变量（ Condition Variable）或者条件队列（ Condition Queue），每个 Condition实例内部都维护了一个用于存储等待线程的队列（等待队列）

- 设 cond1和 cond2是两个不同的 Condition实例，一个线程执行 cond1. await()会导致其被暂停（线程生命周期状态变更为 WAITING）并被存入 cond1的等待队列。 cond1. signal()会使 cond1的等待队列中的一个任意线程被唤醒。 cond1. signalAll()会使 cond1的等待队列中的所有线程被唤醒，而 cond2的等待队列中的任何一个等待线程不受此影响。

BlockingQueue

- A blocking queue causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue.
- The BlockingQueue interface extends java.util.Queue and provides the synchronized put and take methods for adding an element to the head of the queue and for removing an element from the tail of the queue

ConsumerProducerUsingBlockingQueue.java

Semaphores

A thread accessing a shared resource

Acquire a permit from a semaphore.
Wait if the permit is not available.

```
semaphore.acquire();
```

Access the resource

Release the permit to the semaphore

```
semaphore.release();
```

java.util.concurrent.Semaphore

```
+Semaphore(numberOfPermits: int)

+Semaphore(numberOfPermits: int, fair:
  boolean)
+acquire(): void

+release(): void
```