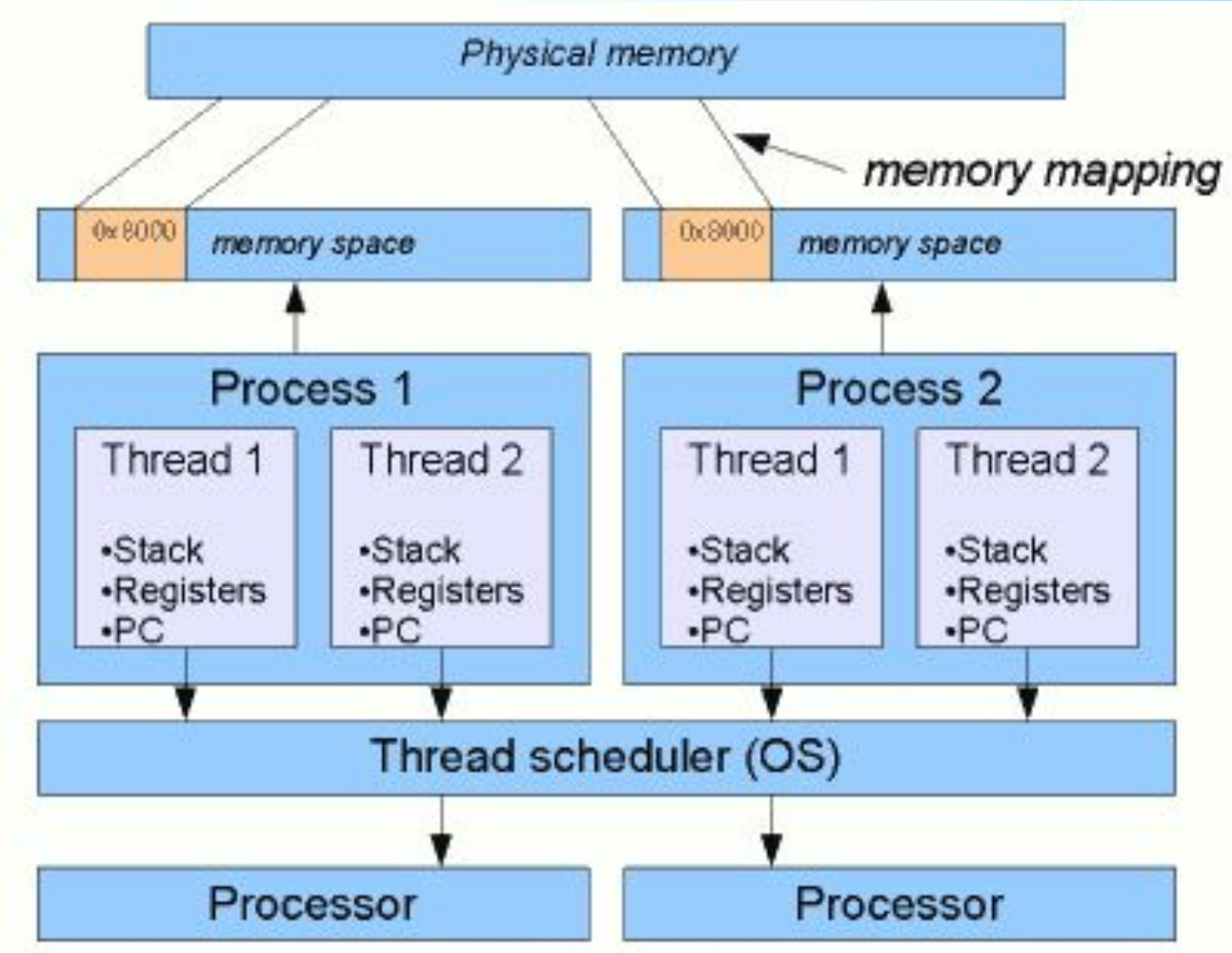
SIMBIRSOFT

Многопоточность


Physical memory
memory mapping
0x8000
memory space
0x8000
memory space
Process 1
Thread 1
•Stack
•Registers
•PC
Thread 2
•Stack
•Registers
•PC
Process 2
Thread 1
•Stack
•Registers
•PC
Thread 2
•Stack
•Registers
•PC
Thread scheduler (OS)
Processor
Processor

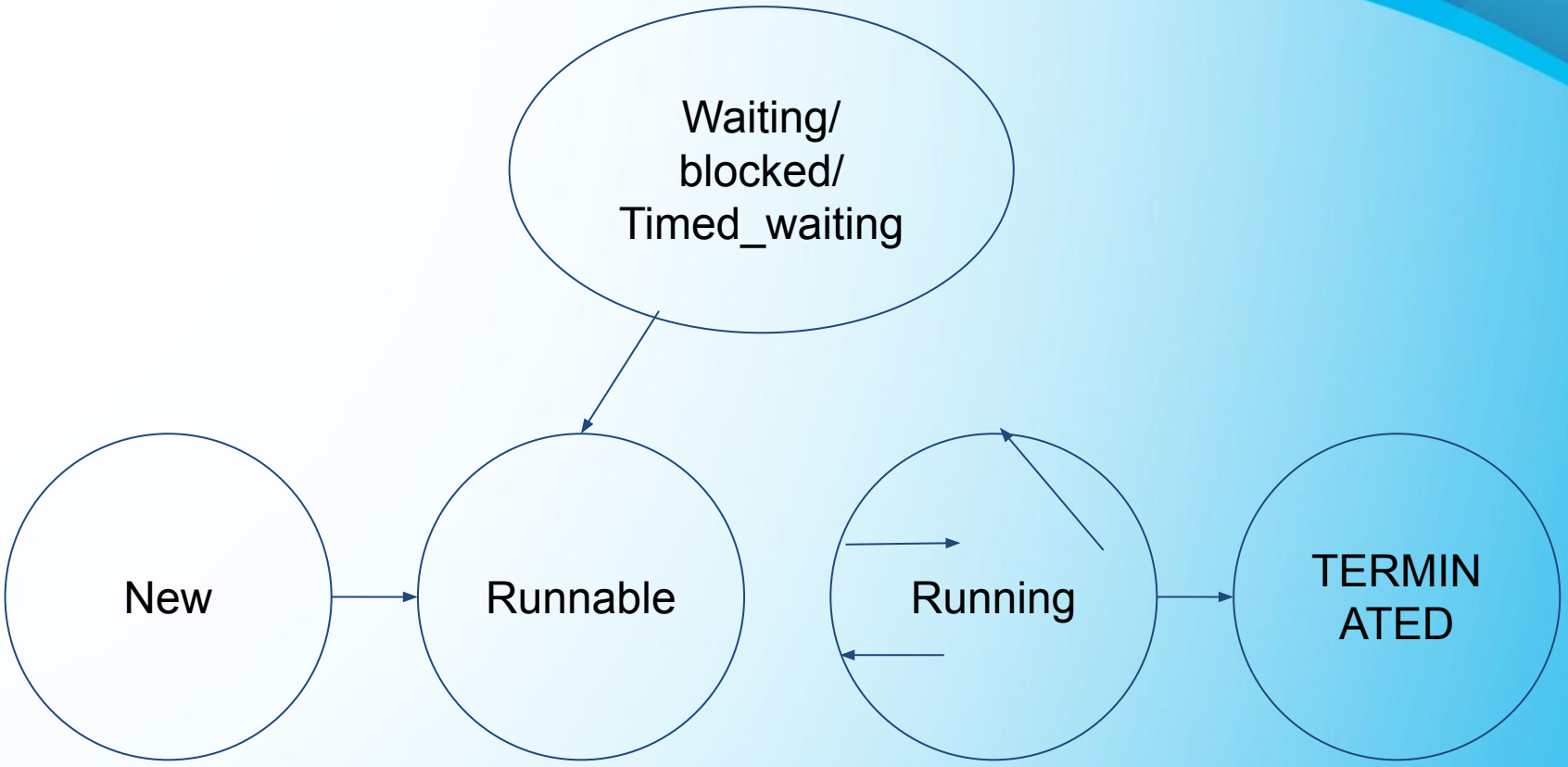
Java Concurrency

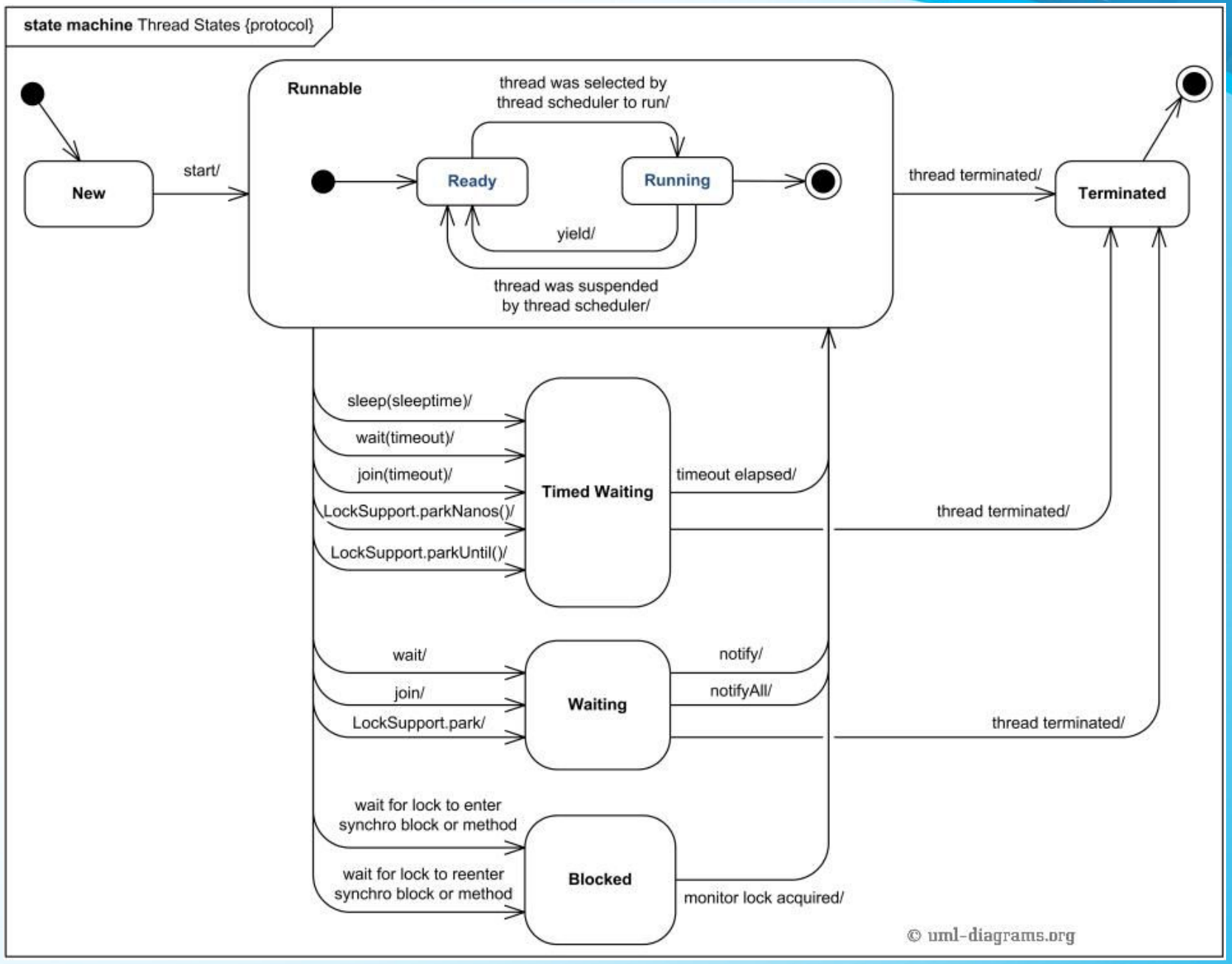
2 подхода к разработке многопоточных приложений:

Низкоуровневый: **`Thread, Runnable, wait/notify, synchronized`**.

1. Пакет **`java.util.concurrent`**: высокоуровневое API параллельного программирования

Состояния потока




state machine Thread States {protocol}
New
start/
Runnable
thread was selected by
thread scheduler to run/
Ready
Running
yield/
thread was suspended
by thread scheduler/
thread terminated/
Terminated
sleep(sleeptime)/
wait(timeout)/
join(timeout)/
LockSupport.parkNanos()/
LockSupport.parkUntil()/
Timed Waiting
timeout elapsed/
thread terminated/
wait/
join/
LockSupport.park/
Waiting
notify/
notifyAll/
thread terminated/
wait for lock to enter
synchro block or method
wait for lock to reenter
synchro block or method
Blocked
monitor lock acquired/
© uml-diagrams.org

Класс Thread

- **getName()** - получить имя потока;
- **getPriority()** - получить приоритет потока;
- **isAlive()** - определить, выполняется ли поток;
- **join()** - ожидать завершения потока;
- **run()** - метод, содержащий код, который выполняется в данном потоке;
- **start()** - запустить поток;
- [static] **sleep()** - приостановить выполнение текущего потока на заданное время.

Создание потока: наследование Thread

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hello");
        }
    }
}
```

Запуск потока:

```
new MyThread().start();
```

Создание потока: реализация Runnable

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("world");
        }
    }
}
```

Запуск потока:

```
new Thread(new MyRunnable()).start();
```

Синхронизация

```
class Counter {
   private int count;

   public void increment() { count++; }

   public int getValue() { return count; }
}
```

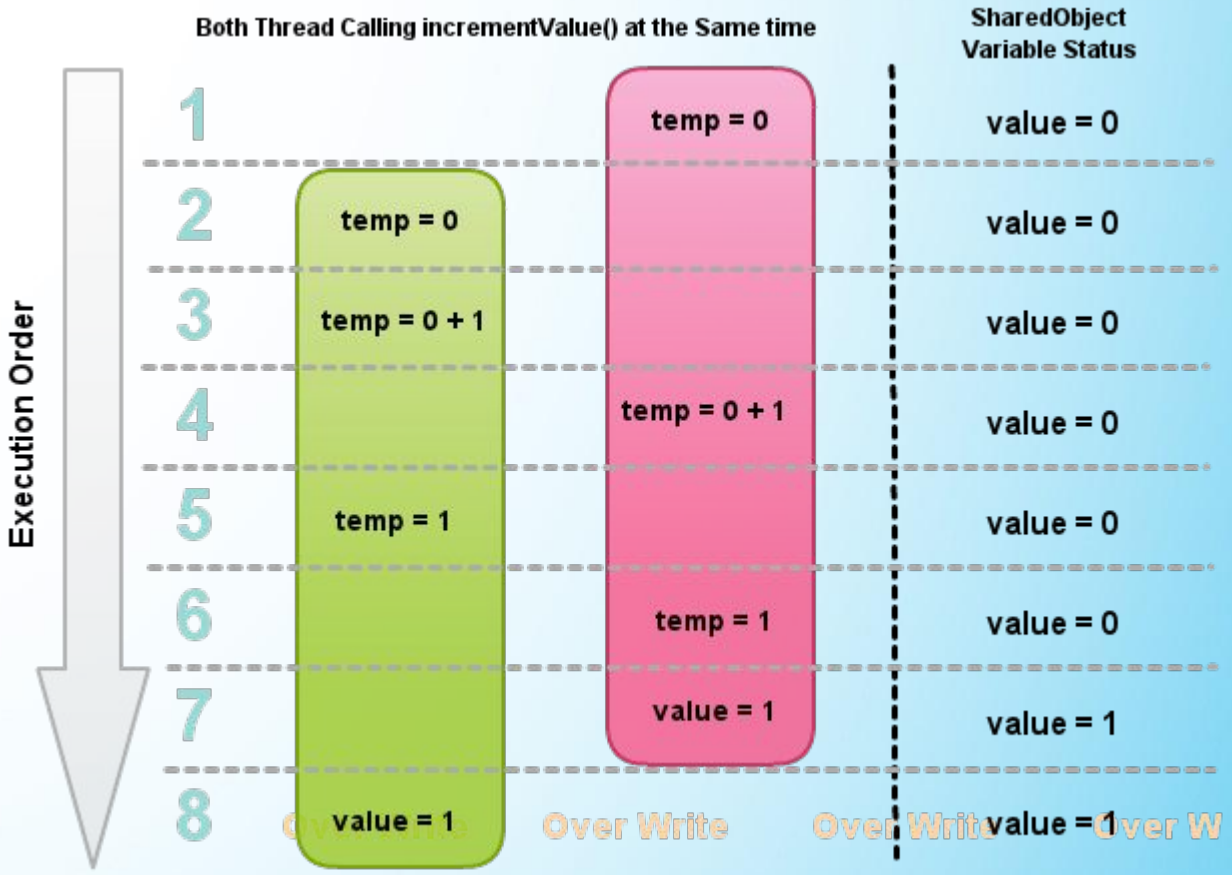
```
class IncrementerThread extends Thread {
   private Counter counter;

   public IncrementerThread(Counter counter) {  this.counter = counter; }

   public static void runExperiment(Counter counter)  throws Exception {
       Thread t1 = new IncrementerThread(counter);
       Thread t2 = new IncrementerThread(counter);
       long startTime = System.currentTimeMillis();
       t1.start(); t2.start();
       t1.join();  t2.join();
       long elapsed = System.currentTimeMillis() - startTime;
       System.out.println("counter=" + counter.getValue() +  ", time
elapsed(ms)=" + elapsed);
   }

   public void run() {
       for (int i = 0; i < 100_000_000; i++) {
           counter.increment();
       }
   }
}
```

Результаты UnsafeCounter

counter = 100 020 579, time elapsed(ms)= 36
counter = 106 287 016, time elapsed(ms)= 33
counter = 105 950 712, time elapsed(ms)= 32
counter = 101 197 861, time elapsed(ms)= 32
counter = 100 029 825, time elapsed(ms)= 40

avg = 34.6 мс

Очень быстро, но неверно (ожидали 200 000 000).


Both Thread Calling incrementValue() at the Same time
SharedObject
Variable Status
Execution Order
1
2
3
4
5
6
7
8
temp = 0
temp = 0
temp = 0 + 1
temp = 1
value = 1
temp = 0
temp = 0 + 1
temp = 1
value = 1
value = 1
Over Write
Over Write
Over Write
Over W
value = 0
value = 0
value = 0
value = 0
value = 0
value = 0
value = 1
value = 1

Синхронизированные методы

```
public class SafeCounterSynchronized implements Counter {
    private int count;

    @Override
    public synchronized void increment() {
        count++;
    }

    @Override
    public synchronized int getValue() {
        return count;
    }
}
```

avg time = 29,6 c

Пример ReentrantLock

Взаимодействие между потоками

```
public class JobQueue
{
 ArrayList<Runnable> jobs = new ArrayList<Runnable>();

 public synchronized void put(Runnable job)
 {
  jobs.add(job);
  this.notifyAll();
 }

 public synchronized Runnable getJob()
 {
  while (jobs.size()==0)
   this.wait();

  return jobs.remove(0);
 }
}
```

wait() / notify() / notifyAll()

```
class DataManager {
    private static Object monitor = new Object();
    public void sendData() {
        synchronized (monitor) {
            try {
                while (!someCondition) monitor.wait();
            }
            catch (InterruptedException ex) {}
            System.out.println("Sending data...");
        }
    }
    public void prepareData() {
        synchronized (monitor) {
            System.out.println("Data is ready");
            monitor.notifyAll();
        }
    }
}
```

Deadlock

Deadlock

```
public class DeadlockRisk {
    private static class Resource {public int value;}
    private Resource resourceA = new Resource();
    private Resource resourceB = new Resource();

    public int read() {
        synchronized (resourceA) {
            synchronized (resourceB) {
                return resourceA.value + resourceB.value;
            }
        }
    }

    public void write (int a, int b) {
        synchronized (resourceB) {
            synchronized (resourceA) {
                resourceA.value = a;
                resourceB.value = b;
            }
        }
    }
}
```

Атомарные операции

Атомарные операции выполняются целиком, их выполнение не может быть прервано планировщиком потоков.

Специальные классы для выполнения атомарных операций находятся в пакете **`java.util.concurrent.atomic`**:

`AtomicInteger`
`AtomicLong`
`AtomicDouble`
`AtomicReference`

… и другие.

Пример: AtomicInteger

```
import java.util.concurrent.atomic.AtomicInteger;

class SafeCounterAtomic implements Counter {
    private AtomicInteger count = new AtomicInteger(0);

    @Override
    public void increment() {
        count.incrementAndGet();
    }

    @Override
    public int getValue() {
        return count.intValue();
    }
}
```

avg time = 13,2 c

interface Lock

Находится в пакете java.util.concurrent.locks

В отличие от synchronized Lock
является не средством языка,
а обычным объектом с набором методов.
В этом случае критическую секцию
ограничивают операции `lock()` и `unlock()`
Вызов lock() блокирует, если Lock в данный
момент занят, поэтому удобно использовать метод
`tryLock()`, который сразу вернет управление
и результат

```
public class Counter{

  private Lock lock = new ReentrantLock();
  private int count = 0;

  public int increment(){
    lock.lock();
    int newCount = ++count;
    lock.unlock();
    return newCount;
  }
}
```

- При использовании Lock не будут работать стандартные методы `wait()`, `notify()` и `notifyAll()`, ведь монитор как таковой не используется
- Вместо них используются реализации интерфейса Condition, ассоциированные с Lock: необходимо вызвать `Lock.newCondition()` и уже у Condition вызывать методы `await()`, `signal()` и `signalAll()`

С одним Lock можно ассоциировать несколько Condition

class ReentrantLock

interface Lock

- Наиболее распространенный паттерн для работы с Lock'ами представлен справа

```
Lock l =...;
l.lock();
try {
    // работаем с ресурсом, который
    // защищен этим мьютексом
} finally {
    l.unlock();
}
```

- Он гарантирует, что Lock будет отпущен в любом случае, даже если при работе с ресурсом будет выброшено исключение
- Для synchronized этот подход неактуален – там средствами языка предоставляется гарантия, что мьютекс будет отпущен
- Этот паттерн весьма полезен в любой ситуации, требующей обязательного освобождения ресурсов
- Широко используются две основные реализации Lock:
- ReentrantLock допускает вложенные критические секции
- ReadWriteLock имеет разные механизмы блокировки на чтение и запись, позволяя уменьшить накладные расходы

Пример ReentrantLock

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class SafeCounterWithLocks implements Counter {
    private int count;
    private Lock lock = new ReentrantLock();

    @Override
    public void increment() {
        lock.lock();
        count++;
        lock.unlock();
    }

    @Override
    public int getValue() {
        lock.lock();
        int value = count;
        lock.unlock();
        return value;
    }
}
```

avg time = 22,4 c

Пример tryLock()

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class SafeCounterWithTryLock implements Counter {
   private int count;
   private Lock lock = new ReentrantLock();

   @Override
   public void increment() {
       boolean locked = false;
       while (!locked) {
           locked = lock.tryLock();
       }
       if (locked) {
           count++;
           lock.unlock();
       }
   }

   @Override
   public int getValue() {
       lock.lock();
       int value = count;
       lock.unlock();
       return value;
   }
}
```


avg time = 32.6 c

interface ReadWriteLock

Методы:

```
Lock readLock();
Lock writeLock();
```

Читать могут несколько потоков одновременно.
Но писать может только один поток.

```
class ReentrantReadWriteLock
```

Пример ReadWriteLock

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class SafeCounterReentrantRWLock  implements Counter {
    private int count;
    private ReadWriteLock  lock = new ReentrantReadWriteLock();

    @Override
    public void increment() {
        lock.writeLock().lock();
        count++;
        lock.writeLock().unlock();
    }

    @Override
    public int getValue() {
        lock.readLock().lock();
        int value = count;
        lock.readLock().unlock();
        return value;
    }
}
```

avg time = 21,8 c

Сравнение производительности

Метод синхронизации	Время выполнения (с)
atomic	13.2
ReentrantReadWriteLock	21.8
ReentrantLock	22.4
synchronized	29.6
tryLock() + while	32.6

interface Condition

Методы:

```
void await() throws InterruptedException;
void signal();
void signalAll();
```

Создание:

```
Lock lock = new ReentrantLock();
Condition blockingPoolA = lock.newCondition();
Condition blockingPoolB = lock.newCondition();
Condition blockingPoolC = lock.newCondition();
```

Condition: применение

Первый поток захватывает блокировку, затем вызывает await() у объекта Condition:

```
lock.lock();
try { blockingPoolA.await(); // ждём второй поток
// продолжаем работу
}
catch (InterruptedException ex) {}
finally {lock.unlock();}
```

Второй поток выполняет свою часть и будит первый поток:

```
lock.lock();
try {
// выполнение работы
blockingPoolA.signalAll(); //будим 1 поток
}
finally {lock.unlock();}
```

Concurrent Collections

- CopyOnWriteArrayList
- CopyOnWriteArraySet

- ConcurrentHashMap
- ConcurrentLinkedDeque
- ConcurrentLinkedQueue
- ConcurrentSkipListMap
- ConcurrentSkipListSet

Copy-on-write

- CopyOnWriteArrayList и CopyOnWriteArraySet основаны на массиве, копируемом при операции записи
- Уже открытые итераторы при этом не увидят изменений в коллекции
- Эти коллекции следует использовать только когда 90+% операций являются операциями чтения
- При частых операциях модификации большая коллекция способна убить производительность
- Сортировка этих коллекций не поддерживается, т.к. она подразумевает O(n) операций вставки
- Итераторы по этим коллекциям не поддерживают операций модификации

Синхронизаторы

- Предназначены для регулирования и ограничения потоков. предоставляют более высокий уровень абстракции, чем мониторы.
- Semaphore
- CountDownLatch
- CyclicBarrier

Semaphore

- Объект, позволяющий войти в заданный участок кода не более чем *n* потокам одновременно
- N определяется параметром конструктора
- При N=1 по действию аналогичен Lock
- Fairness – гарантия очередности потоков

```
Semaphore semaphore = new Semaphore(4);
semaphore.acquire();
semaphore.release();
boolean available = semaphore.tryAcquire();
```

Блокирующие очереди

ArrayBlockingQueue
LinkedBlockingDeque
LinkedBlockingQueue

bounded

PriorityBlockingQueue
DelayQueue — *элементы с задержкой*
LinkedTransferQueue — *универсальная очередь*
SynchronousQueue — *ёмкость 0*

Bounded Queues: пример

```
BlockingQueue<Integer> bq = new ArrayBlockingQueue<>(1);
try {
    bq.put(24);
    bq.put(25); // блокировка до удаления 24 из очереди
} catch (InterruptedException ex) {
}
```

Блокирующие очереди: методы

Методы получения элементов блокирующей очереди:

take() - возвращает первый объект очереди, удаляя его из очереди.
Если очередь пустая, блокируется.

poll() - возвращает первый объект очереди, удаляя его из очереди.
Если очередь пустая, возвращает null.

element() - возвращает первый элемент очереди, не удаляя его из очереди.
Если очередь пустая, то NoSuchElementException.

peek() - возвращает первый элемент очереди, не удаляя его из очереди.
Если очередь пустая, возвращает null.

java.util.concurrent.Executor

Цель применения: отделить работу, выполняемую внутри потока, от логики создания потоков.

Создание:

```
public class SimpleThreadExecutor implements Executor {
    @Override
    public void execute(Runnable command) {
        command.run();
    }
}
```

Использование:

```
Runnable runnable = new MyRunnableTask();
Executor executor = new SimpleThreadExecutor();
executor.execute(runnable);
```

Стандартные Executor-ы

```
Executors.newCachedThreadPool();
```
создаёт новые потоки при необходимости, повторно использует освободившиеся потоки

```
Executors.newFixedThreadPool(12);
```
с ограничением количества потоков

```
Executors.newSingleThreadExecutor();
```
ровно один поток

```
Executors.newScheduledThreadPool();
```
можно настроить задержку запуска / повторный запуск

Все эти методы возвращают ExecutorService:

```
public interface ExecutorService extends Executor
```

java.util.concurrent.Callable

В `ExecutorService` можно передавать Callable и Runnable.

Разница: Callable может возвращать результат (в виде Future).

```
class MyCallable implements Callable<String> {
    @Override
    public String call() {
          StringBuilder builder = new StringBuilder(str);
          builder.reverse();
          return builder.toString();
    }
}
```

Future

```
Callable<String> callable = new MyCallable();
ExecutorService ex = Executors.newCachedThreadPool();
Future<String> f = ex.submit(callable);

try {
    Integer v = f.get();
    System.out.println(v);
} catch (InterruptedException | ExecutionException e) {}
```

boolean **cancel**(boolean mayInterrupt);	Останавливает задачу.
boolean **isCancelled**();	Возвращает true, если задача была остановлена.
boolean **isDone**();	Возвращает true, если выполнение задачи завершено.
get();	Возвращает результат вызова метода call или кидает исключение, если оно было.