**Synchronizers**

*Semaphore*

**import** java.util.concurrent.Semaphore;

**public** **class** SemaphoreExample {

**public** **static** **void** main(String[] args) {

Semaphore semaphore = **new** Semaphore(1);

**new** Worker(semaphore, "Adder", **true**).start(); // стартуем потоки

**new** Worker(semaphore, "Reducer", **false**).start();

}

}

**public** **class** Cart {

// класс тележка

**private** **static** **int** *weight* = 0;

**public** **static** **void** addWeight(){

*weight*--;

}

**public** **static** **void** reduceWeight(){

*weight*++;

}

**public** **static** **int** getWaight(){

**return** *weight*;

}

}

**import** java.util.concurrent.Semaphore;

**public** **class** Worker **extends** Thread {

// класс работника (boolean переменная определяет это работник по загрузке тележки либо по разгрузке)

**private** Semaphore semaphore;

**private** String workerName;

**private** **boolean** isAdder;

**public** Worker(Semaphore semaphore, String workerName, **boolean** isAdder) {

**this**.semaphore = semaphore;

**this**.workerName = workerName;

**this**.isAdder = isAdder;

}

@Override

**public** **void** run() {

System.***out***.println(workerName + " started working...");

**try** {

System.***out***.println(workerName + " waiting for cart...");

semaphore.acquire();

System.***out***.println(workerName + " got access to cart...");

**for** (**int** i = 0 ; i < 10 ; i++) {

**if** (isAdder)

Cart.*reduceWeight*();

**else**

Cart.*addWeight*();

System.***out***.println(workerName + " changed weight to: " + Cart.*getWaight*());

Thread.*sleep*(10L);

}

System.***out***.println(workerName + " finished working with cart...");

} **catch** (Exception e) {

e.printStackTrace(System.***err***);

} **finally** {

semaphore.release();

}

}

}

*CountDownLatch*

**import** com.google.common.base.Stopwatch;

**import** java.util.concurrent.CountDownLatch;

**import** java.util.concurrent.TimeUnit;

**public** **class** Main {

// главный класс, где создаются работники и защелка на 4 события. Этот класс ожидает завершения работы всех работников (latch.await())

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Stopwatch sw = **new** Stopwatch();

sw.start();

CountDownLatch latch = **new** CountDownLatch(4);

**new** Worker(latch,"Sand").start();

**new** Worker(latch,"Cement").start();

**new** Worker(latch,"Water").start();

**new** Worker(latch,"Breakstone").start();

System.out.println("Waiting for all workers");

latch.await();

System.out.println("All workers finished. Now we can shake.");

sw.stop();

System.out.println("Program time: " + sw.elapsed(TimeUnit.MILLISECONDS) + " ms");

}

}

**public** **class** RandomGenerator {

// Класс для генерации случайного времени задержки (время условно определяет удаленность материала от емкости)

**public** **static** **long** getRandom(**long** min, **long** max) {

**return** min + (**long**)(Math.*random*() \* (max - min + 1));

}

}

**import** com.google.common.base.Stopwatch;

**import** java.util.concurrent.CountDownLatch;

**import** java.util.concurrent.TimeUnit;

**public** **class** Worker **extends** Thread {

// класс работника, который выполняет задачу и извещает о выполнении

**private** CountDownLatch \_cdl;

**private** String \_name;

**public** Worker (CountDownLatch cdl, String name) {

\_cdl = cdl;

\_name = name;

}

@Override

**public** **void** run() {

Stopwatch sw = **new** Stopwatch();

sw.start();

System.***out***.println(\_name + " working...");

**try** {

Thread.*sleep*(RandomGenerator.*getRandom*(500, 1000));

} **catch** (InterruptedException e) {

e.printStackTrace(System.***err***);

}

\_cdl.countDown();

sw.stop();

System.***out***.println(\_name + " time: " + sw.elapsed(TimeUnit.***MILLISECONDS***) + " ms");

}

}

*CyclicBarrier*

**import** java.util.concurrent.BrokenBarrierException;

**import** java.util.concurrent.CyclicBarrier;

**import** java.util.logging.Level;

**import** java.util.logging.Logger;

**public** **class** CyclicBarrierExample {

//Runnable для каждого потока

**private** **static** **class** Task **implements** Runnable {

**private** CyclicBarrier barrier;

**public** Task(CyclicBarrier barrier) {

**this**.barrier = barrier;

}

@Override

**public** **void** run() {

**try** {

System.***out***.println(Thread.*currentThread*().getName() + " is waiting on barrier");

barrier.await();

System.***out***.println(Thread.*currentThread*().getName() + " has crossed the barrier");

} **catch** (InterruptedException ex) {

Logger.*getLogger*(CyclicBarrierExample.**class**.getName()).log(Level.***SEVERE***, **null**, ex);

} **catch** (BrokenBarrierException ex) {

Logger.*getLogger*(CyclicBarrierExample.**class**.getName()).log(Level.***SEVERE***, **null**, ex);

}

}

}

**public** **static** **void** main(String args[]) {

//создание CyclicBarrier 3 потока нуждаются в вызове await()

**final** CyclicBarrier cb = **new** CyclicBarrier(3, **new** Runnable(){

@Override

**public** **void** run(){

// Эта задача будет выполняться один раз, когда все потоки достигнут Barrier

System.***out***.println("All parties are arrived at barrier, lets play");

}

});

//starting each of thread

Thread t1 = **new** Thread(**new** Task(cb), "Thread 1");

Thread t2 = **new** Thread(**new** Task(cb), "Thread 2");

Thread t3 = **new** Thread(**new** Task(cb), "Thread 3");

t1.start();

t2.start();

t3.start();

}

}

*Exchanger*

**import** java.util.concurrent.Exchanger;

**public** **class** Delivery {

//Создаем обменник, который будет обмениваться типом String

**private** **static** **final** Exchanger<String> ***EXCHANGER*** = **new** Exchanger<>();

**public** **static** **void** main(String[] args) **throws** InterruptedException {

String[] p1 = **new** String[]{"{посылка A->D}", "{посылка A->C}"};//Формируем груз для 1-го грузовика

String[] p2 = **new** String[]{"{посылка B->C}", "{посылка B->D}"};//Формируем груз для 2-го грузовика

**new** Thread(**new** Truck(1, "A", "D", p1)).start();//Отправляем 1-й грузовик из А в D

Thread.*sleep*(100);

**new** Thread(**new** Truck(2, "B", "C", p2)).start();//Отправляем 2-й грузовик из В в С

}

**public** **static** **class** Truck **implements** Runnable {

**private** **int** number;

**private** String dep;

**private** String dest;

**private** String[] parcels;

**public** Truck(**int** number, String departure, String destination, String[] parcels) {

**this**.number = number;

**this**.dep = departure;

**this**.dest = destination;

**this**.parcels = parcels;

}

@Override

**public** **void** run() {

**try** {

System.***out***.printf("В грузовик №%d погрузили: %s и %s.\n", number, parcels[0], parcels[1]);

System.***out***.printf("Грузовик №%d выехал из пункта %s в пункт %s.\n", number, dep, dest);

Thread.*sleep*(1000 + (**long**) Math.*random*() \* 5000);

System.***out***.printf("Грузовик №%d приехал в пункт Е.\n", number);

parcels[1] = ***EXCHANGER***.exchange(parcels[1]);//При вызове exchange() поток блокируется и ждет

//пока другой поток вызовет exchange(), после этого произойдет обмен посылками

System.***out***.printf("В грузовик №%d переместили посылку для пункта %s.\n", number, dest);

Thread.*sleep*(1000 + (**long**) Math.*random*() \* 5000);

System.***out***.printf("Грузовик №%d приехал в %s и доставил: %s и %s.\n", number, dest, parcels[0], parcels[1]);

} **catch** (InterruptedException e) {

}

}

}

}

*Phaser*

**import** java.util.ArrayList;

**import** java.util.concurrent.Phaser;

**public** **class** Bus {

**private** **static** **final** Phaser ***PHASER*** = **new** Phaser(1);//Сразу регистрируем главный поток

//Фазы 0 и 6 - это автобусный парк, 1 - 5 остановки

**public** **static** **void** main(String[] args) **throws** InterruptedException {

ArrayList<Passenger> passengers = **new** ArrayList<>();

**for** (**int** i = 1; i < 5; i++) { //Сгенерируем пассажиров на остановках

**if** ((**int**) (Math.*random*() \* 2) > 0)

passengers.add(**new** Passenger(i, i + 1));//Этот пассажир выходит на следующей

**if** ((**int**) (Math.*random*() \* 2) > 0)

passengers.add(**new** Passenger(i, 5)); //Этот пассажир выходит на конечной

}

**for** (**int** i = 0; i < 7; i++) {

**switch** (i) {

**case** 0:

System.***out***.println("Автобус выехал из парка.");

***PHASER***.arrive();//В фазе 0 всего 1 участник - автобус

**break**;

**case** 6:

System.***out***.println("Автобус уехал в парк.");

***PHASER***.arriveAndDeregister();//Снимаем главный поток, ломаем барьер

**break**;

**default**:

**int** currentBusStop = ***PHASER***.getPhase();

System.***out***.println("Остановка № " + currentBusStop);

**for** (Passenger p : passengers) //Проверяем, есть ли пассажиры на остановке

**if** (p.departure == currentBusStop) {

***PHASER***.register();//Регистрируем поток, который будет участвовать в фазах

p.start(); // и запускаем

}

***PHASER***.arriveAndAwaitAdvance();//Сообщаем о своей готовности

}

}

}

**public** **static** **class** Passenger **extends** Thread {

**private** **int** departure;

**private** **int** destination;

**public** Passenger(**int** departure, **int** destination) {

**this**.departure = departure;

**this**.destination = destination;

System.***out***.println(**this** + " ждёт на остановке № " + **this**.departure);

}

@Override

**public** **void** run() {

**try** {

System.***out***.println(**this** + " сел в автобус.");

**while** (***PHASER***.getPhase() < destination) //Пока автобус не приедет на нужную остановку(фазу)

***PHASER***.arriveAndAwaitAdvance(); //заявляем в каждой фазе о готовности и ждем

Thread.*sleep*(1);

System.***out***.println(**this** + " покинул автобус.");

***PHASER***.arriveAndDeregister(); //Отменяем регистрацию на нужной фазе

} **catch** (InterruptedException e) {

}

}

@Override

**public** String toString() {

**return** "Пассажир{" + departure + " -> " + destination + '}';

}

}

}

**Locks**

*Condition*

**public** **class** ConditionExample {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

SharedFiFoQueue sharedQueue = **new** SharedFiFoQueue(10);

//Создание producer и consumer.

Thread producer = **new** Producer(sharedQueue);

Thread consumer = **new** Consumer(sharedQueue);

//Старт обоих потоков.

producer.start();

consumer.start();

//Ожидание выполнения обоих потоков.

producer.join();

consumer.join();

}

}

*Lock, ReentrantLock*

**import** java.util.concurrent.locks.ReentrantLock;

**public** **class** ThreadsApp {

**public** **static** **void** main(String[] args) {

CommonResource commonResource= **new** CommonResource();

ReentrantLock locker = **new** ReentrantLock(); // создаем заглушку

**for** (**int** i = 1; i < 6; i++){

Thread t = **new** Thread(**new** CountThread(commonResource, locker));

t.setName("Поток "+ i);

t.start();

}

}

}

**class** CommonResource{

**int** x=0;

}

**class** CountThread **implements** Runnable{

CommonResource res;

ReentrantLock locker;

CountThread(CommonResource res, ReentrantLock lock){

**this**.res=res;

locker = lock;

}

**public** **void** run(){

**try**{

locker.lock(); // устанавливаем блокировку

res.x=1;

**for** (**int** i = 1; i < 5; i++){

System.out.printf("%s %d \n", Thread.currentThread().getName(), res.x);

res.x++;

Thread.sleep(100);

}

}

**catch**(InterruptedException e){

System.out.println(e.getMessage());

}

**finally**{

locker.unlock(); // снимаем блокировку

}

}

}

*ReadWriteLock, ReentrantReadWriteLock*

**import** java.util.Map;

**import** java.util.TreeMap;

**import** java.util.concurrent.ConcurrentHashMap;

**import** java.util.concurrent.locks.ReentrantReadWriteLock;

**import** java.util.concurrent.locks.ReadWriteLock;

**public** **class** ThreadsafeTrie {

ReadWriteLock rwlLock = **new** ReentrantReadWriteLock();

**static** **class** TrieNode {

Map<Character, TrieNode> children = **new** TreeMap<Character, TrieNode>();

**boolean** leaf;

}

TrieNode root = **new** TrieNode();

**public** **void** put(String s) {

TrieNode v = root;

rwlLock.writeLock().lock();

{

**for** (**char** ch : s.toLowerCase().toCharArray()) {

**if** (!v.children.containsKey(ch)) {

v.children.put(ch, **new** TrieNode());

}

v = v.children.get(ch);

}

v.leaf = **true**;

}

rwlLock.writeLock().unlock();

}

**public** **boolean** find(String s) {

TrieNode v = root;

**boolean** result = **true**;

rwlLock.readLock().lock();

{

**for** (**char** ch : s.toLowerCase().toCharArray()) {

**if** (!v.children.containsKey(ch)) {

result = **false**;

**break**;

} **else** {

v = v.children.get(ch);

}

}

}

rwlLock.readLock().unlock();

**return** result;

}

**static** Map<Integer,String> *levelSpacesMap* = **new** ConcurrentHashMap<Integer,String>();

**static** String getSpace(**int** level) {

String result = *levelSpacesMap*.get(level);

**if** (result == **null**) {

StringBuilder sb = **new** StringBuilder();

**for** (**int** i=0; i<level; i++) {

sb.append(" ");

}

result = sb.toString();

*levelSpacesMap*.put(level,result);

}

**return** result;

}

**public** **void** printSorted() {

rwlLock.readLock().lock();

{

printSorted2(root,0);

}

rwlLock.readLock().unlock();

}

**private** **void** printSorted2(TrieNode node, **int** level) {

**for** (Character ch : node.children.keySet()) {

System.***out***.println(*getSpace*(level)+ch);

printSorted2(node.children.get(ch), level+1);

}

**if** (node.leaf) {

System.***out***.println();

}

}

**public** **static** **void** main(String[] args) {

ThreadsafeTrie trie = **new** ThreadsafeTrie();

trie.put("hello");

trie.put("house");

trie.put("hell");

trie.put("world");

System.***out***.println(trie.find("hello"));

trie.printSorted();

}

}