МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ В JAVA

- Пакеты
  - java.lang
  - java.util.concurrent

CONCURRENCY UTILITIES

- Набор классов, облегчающих написание многопоточных программ
- Пакет java.util.concurrent.locks
  - Работа с блокировками
- Пакет java.util.concurrent.atomic
  - Атомарные переменные
- Пакет java.util.concurrent
  - Примитивы синхронизации
  - Многопоточные коллекции
  - Управление заданиями

БЛОКИРОВКИ И УСЛОВИЯ

ЧАСТЬ 1

БЛОКИРОВКА

- Только один поток может владеть блокировкой
- Операции
  - lock — получить блокировку
  - unlock — отдать блокировку
  - tryLock — попробовать получить блокировку

БЛОКИРОВКИ В JAVA

- Интерфейс Lock
- Методы
  - lock() – захватить блокировку
  - lockInterruptibly() – захватить блокировку
  - tryLock(time?) – попытаться захватить блокировку
  - unlock() – отпустить блокировку
- Передача событий
  - newCondition() – создать условие

Условия

- Интерфейс Condition
  - await(time?) – ждать условия
  - awaitUntil(deadline) – ждать условия до времени
  - awaitUninterruptibly() – ждать условие
  - signal() – подать сигнал
  - signalAll() – подать сигнал всем
- Нужно владеть родительской блокировкой

ПРОИЗВОДИТЕЛЬ

- Решение с помощью событий

```
void set(Object data) throws InterruptedException {
    lock.lock();
    try {
        while (data != null) notFull.await();
        this.data = data;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

ОСОБЕННОСТИ

- Отсутствие «блочности»
  - Разделенные блокировки
- Необходимость явного отпускания
- Идиома

```
l.lock()
try {
    …
} finally {
    l.unlock()
}
```

РЕАЛИЗАЦИЯ БЛОКИРОВКИ

- Класс ReentrantLock
- Дополнительные методы
  - isFair() – «честность» блокировки
  - isLocked() – блокировка занята
- Статистика
  - getQueuedThreads() / getQueueLength() / hasQueuedThread(thread) / hasQueuedThreads() – потоки, ждущие блокировку
  - getWaitingThreads(condition) / getWaitQueueLength(condition) – потоки, ждущие условие

BOUNDED BUFFER

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
            …
   }

   public Object take() throws InterruptedException {
            …
   }
}
```

BOUNDED BUFFER

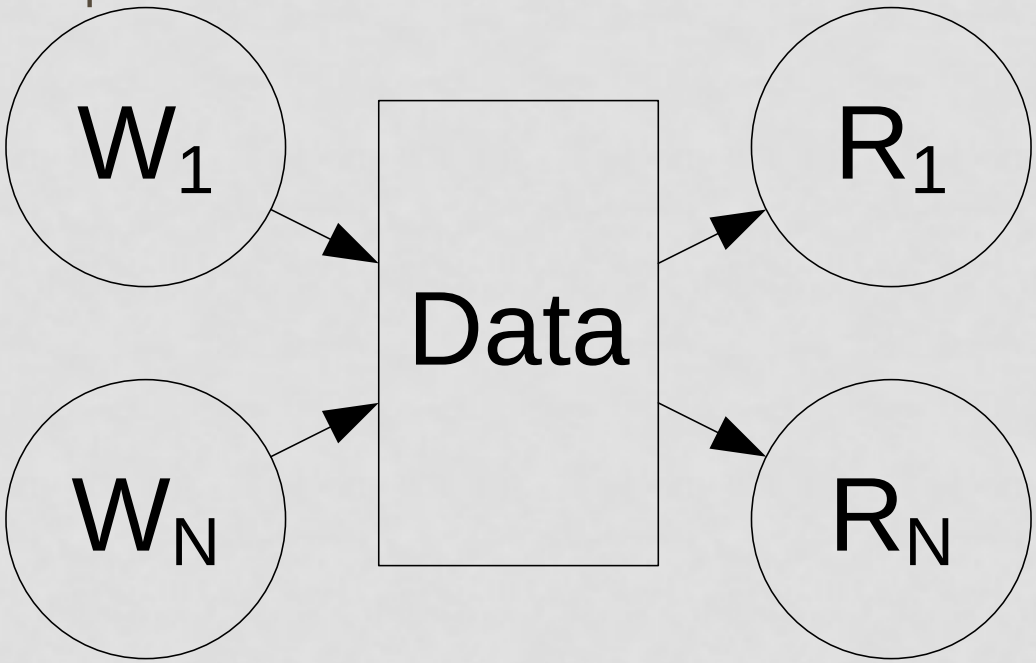
```
public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
```

BOUNDED BUFFER

```
public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      Object x = items[takeptr];
      if (++takeptr == items.length) takeptr = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
```

ЗАДАЧА О ЧИТАТЕЛЯХ И ПИСАТЕЛЯХ

- Читать могут много потоков одновременно
- Писать может только один поток
- Читать во время записи нельзя



ЧИТАТЕЛИ И ПИСАТЕЛИ

- Интерфейс ReadWriteLock
- Методы
  - readLock() – блокировка для читателей
  - writeLock() – блокировка для писателей
- Реализация ReentrantReadWriteLock

УПРАВЛЕНИЕ ЗАДАНИЯМИ

ЧАСТЬ 2

ИСПОЛНИТЕЛИ

- Интерфейс Executor
  - execute(Runnable) – выполнить задание
- Возможные варианты выполнения
  - В том же потоке
  - Во вновь создаваемом потоке
  - Пул потоков
  - Наращиваемый пул потоков

ФУНКЦИИ И РЕЗУЛЬТАТЫ

- Интерфейс Callable<V> – функция
  - V call() – подсчитать функцию
- Интерфейс Future<V> – результат
  - get(timeout?) – получить результат
  - isDone() – окончено ли выполнение
  - cancel(mayInterruptWhenRunning) – прервать выполнение
  - isCancelled() – прервано ли выполнение

ИСПОЛНИТЕЛИ-2

- Интерфейс ExecutorService
  - submit(Runnable) – выполнить задание
  - Future<V> submit(Callable<V>) – выполнить функцию
  - List<Future> invokeAll(List<Callable>) – выполнить все функции
  - Future invokeAny(List<Callable>) – успешно выполнить функцию

ЗАВЕРШЕНИЕ РАБОТЫ

- shutdown() – прекратить прием заданий
- List<Runnable> shutdownNow() – прекратить выполнение
- isShutdown() – прекращен ли прием
- isTerminated() – окончен ли все задания
- awaitTermination(timeout) – ожидание завершения

КЛАСС EXECUTORS

- Создание исполнителей
  - newCachedThreadPool()
  - newFixedThreadPool(n)
  - newSingleThreadExecutor()
- Создание фабрик потоков
  - Класс ThreadFactory
- Создание привилегированных действий и фабрик потоков
  - Наследую права создавшего

ПРИМЕР

```
class Task implements Runnable {
        private int counter;
        public Task(int num) { this.counter = num; }
        public void run() {
                while (counter-- > 0) {
                        System.out.println(Thread.currentThread() + ": " + counter);
                        Thread.yield();
                }
        }
}

public class Main {
        public static void main(String[] args) {
                Random rand = new Random();
                ExecutorService exec = Executors.newFixedThreadPool(2);
                for (int i = 0; i < 5; i++) {
                        exec.execute(new Task(Math.abs(rand.nextInt())%10));
                }
                exec.shutdown();
        }
}
```

ПРИМЕР-2

```
class CallableTask implements Callable<Integer> {
        private int counter;
        private final int number;
        public CallableTask(int num) {
          this.counter = num;
          this.number = num;
        }
        public Integer call() {
          while (counter-- > 0) {
          System.out.println(Thread.currentThread() + ": " + counter);
                    Thread.yield();
          }
          return number;
        }
}
```

ПРИМЕР-2

```
public class MainCallable {
	public static void main(String[] args) {
		ArrayList<Future<Integer>> results = new ArrayList<>();

		ExecutorService exec = Executors.newCachedThreadPool();
		for (int i = 0; i < 5; i++) { results.add(exec.submit(new CallableTask(i)));  }
		exec.shutdown();

		for (Future<Integer> fi : results) {
			try {
				System.out.println(fi.get());
			} catch (InterruptedException | ExecutionException e) {
	e.printStackTrace(); }

			try {
				System.out.println(fi.get(5, TimeUnit.SECONDS));
			} catch (InterruptedException | ExecutionException e | TimeoutException e) {
				e.printStackTrace();
			}
		}
	}
}
```

РЕАЛИЗАЦИЯ ИСПОЛНИТЕЛЕЙ

- Класс ThreadPoolExecutor
  - corePoolSize – минимальное количество потоков
  - maxPoolSize максимальное количество потоков
  - blockingQueue – очередь заданий
  - keepAliveTime – время жизни потока
  - threadFactory – фабрика потоков
  - …

ПРИМЕР

```
class DaemonThreadFactory implements ThreadFactory {
        public Thread newThread(Runnable r) {
                Thread t = new Thread(r);
                t.setDaemon(true);
                return t;
        }
}

public class MainThreadFactory {
        public static void main(String[] args) {
                ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();

                ExecutorService exec = Executors.newCachedThreadPool(new DaemonThreadFactory());
                for (int i = 0; i < 5; i++) { results.add(exec.submit(new CallableTask(i*100))); }
                exec.shutdown();

                for (Future<Integer> fi : results) {
                        try {
                                System.out.println(fi.get());
                        } catch (InterruptedException | ExecutionException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

ОТКЛОНЕНИЕ ЗАДАНИЙ

- Нет свободного потока и места в очереди
- Политики отклонения
  - AbortPolicy – бросить RejectedExecutionException
  - CallerRunsPolicy – исполнить в вызывающем потоке
  - DiscardPolicy – проигноировать
  - DiscardOldestPolicy – заменить самое давнее
- Интерфейс RejectedExecutionHandler

ОТЛОЖЕННОЕ ИСПОЛНЕНИЕ

- Интерфейс ScheduledExecutorService
  - schedule(callable, timeout) – исполнить через timeout
  - schedule(runnable, timeout?) – исполнить через timeout
  - sheduleAtFixedRate(runnable, initialDelay, period) – периодическое исполнение
  - scheduleWithFixedDelay(runnable, initialDelay, delay) – исполнение с равными интервалами
  - Все методы возвращают ScheduledFuture

РЕАЛИЗАЦИЯ ОТЛОЖЕННОГО ИСПОЛНЕНИЯ

- Класс ScheduledThreadPoolExecutor

ПРИМИТИВЫ СИНХРОНИЗАЦИИ

ЧАСТЬ 3

ПРИМИТИВЫ СИНХРОНИЗАЦИИ

- Semaphore – семафор
- CyclicBarrier – многоразовый барьер
- CountDownLatch – защелка
- Exchanger – рандеву

СЕМАФОР

- Хранит количество разрешений на вход
- Операции
  - acquire получить разрешение
  - release добавить разрешение
- Доступ к ограниченными ресурсами

СЕМАФОРЫ В JAVA

- Конструкторы
  - Semaphore(n, fair?) – число разрешений и честность
- Методы
  - acquire(n?) – получить разрешение
  - release(n?) – отдать разрешение
  - tryAquire(n?, time?) – попробовать получить разрешение
  - reducePermits(n) – уменьшить количество разрешений
  - drainPermits() – забрать все разрешения
  - статистика

БАРЬЕР

- Потоки блокируются пока все потоки не прибудут к барьеру
  - Одноразовый
  - Многоразовый
- Операции
  - arrive прибытие к барьеру
- Синхронизация потоков
  - Переход к следующему этапу

БАРЬЕРЫ В JAVA

- Конструкторы
  - CyclicBarrier(n, runnable?) – число потоков и действие на барьере
- Методы
  - await(time?) – барьер.
  - reset() – возвращает барьер в исходное состояние
  - isBroken() – «сломан» ли барьер
  - статистика

ПРИМЕР. СКАЧКИ

```
class Horse implements Runnable {
  private int strides = 0;
  private static Random rand = new Random(47);
  private static CyclicBarrier barrier;
  public Horse(CyclicBarrier b) { barrier = b; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        synchronized(this) {
          strides += rand.nextInt(3); // Produces 0, 1 or 2
        }
        barrier.await();
      }
    } catch(InterruptedException e) {// Приемлемый вариант выхода
    } catch(BrokenBarrierException e) {throw new RuntimeException(e);}
  }
              …
}
```

ПРИМЕР. СКАЧКИ

```
public class HorseRace {
  private List<Horse> horses = new ArrayList<Horse>();
  private ExecutorService exec = Executors.newCachedThreadPool();
  private CyclicBarrier barrier;
  public HorseRace(int nHorses) {
    barrier = new CyclicBarrier(nHorses, new Runnable() {
      public void run() {
        for(Horse horse : horses)
                System.out.println(horse.tracks());
        for(Horse horse : horses)
          if(horse.getStrides() >= FINISH_LINE) {
                  System.out.println(horse + "won!");
            exec.shutdownNow();
            return;
          }
        }
    });
    for(int i = 0; i < nHorses; i++) {
      Horse horse = new Horse(barrier); horses.add(horse); exec.execute(horse);
    }
  }
```

МОНИТОР

- Разделяемые переменные инкапсулированы в мониторе
- Код в мониторе исполняется не более чем одним потоком
- Условия
- Операции с условиями
  - wait — ожидание условия
  - notify — сообщение об условии одному потоку
  - notifyAll — сообщение об условии всем потокам

ЗАЩЕЛКИ

- Ожидание завершения нескольких работ
- Операции
  - countDown() – опускает защелку на единицу
  - await() – ждет спуска защелки

ЗАЩЕЛКИ В JAVA

- Конструктор
  - CountDownLatch(n) – высота защелки
- Методы
  - await(time?) – ждет спуска защелки
  - countDown() – опускает защелку на единицу
  - getCount() – текущая высота защелки
- Применение
  - Инициализация

ПРИМЕР

```
// Часть основной задачи:
class TaskPortion implements Runnable {
            …
  private final CountDownLatch latch;
  TaskPortion(CountDownLatch latch) {
    this.latch = latch;
  }
  public void run() {
    try {
      doWork();
      latch.countDown();
    } catch(InterruptedException ex) {
      // Приемлемый вариант выхода
    }
  }
}
```

```
// Ожидание по объекту CountDownLatch:
class WaitingTask implements Runnable {
            …
  private final CountDownLatch latch;
  WaitingTask(CountDownLatch latch) {
    this.latch = latch;
  }
  public void run() {
    try {
      latch.await();
      doWrk();
      System.out.println("barrier passed");
    } catch(InterruptedException ex) {
      System.out.println(this + " interrupted");
    }
  }
}
```

РАНДЕВУ

- Позволяет потокам синхронно обмениваться объектами
- Конструкторы
  - Exchanger()
- Методы
  - exchange(V x, time?) – обменяться

ПРИМЕР

```
class ExchangerProducer<T> implements Runnable {
  private Generator<T> generator;
  private Exchanger<List<T>> exchanger;
  private List<T> holder;
  ExchangerProducer(Exchanger<List<T>> exchg, Generator<T> gen, List<T> holder) {
    exchanger = exchg;
    generator = gen;
    this.holder = holder;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        for(int i = 0; i < ExchangerDemo.size; i++)
          holder.add(generator.next());
          holder = exchanger.exchange(holder);
      }
    } catch(InterruptedException e) {}
  }
}
```

ПРИМЕР

```
class ExchangerConsumer<T> implements Runnable {
  private Exchanger<List<T>> exchanger;
  private List<T> holder;
  private volatile T value;
  ExchangerConsumer(Exchanger<List<T>> ex, List<T> holder){
    exchanger = ex;
    this.holder = holder;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        holder = exchanger.exchange(holder);
        for(T x : holder) {
          value = x; // Выборка значения
          holder.remove(x);
        }
      }
    } catch(InterruptedException e) {}
    System.out.println("Final value: " + value);
  }
}
```

ПРИМЕР

```
public class ExchangerDemo {
  static int size = 10;
  static int delay = 5; // Секунды
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    Exchanger<List<Fat>> xc = new Exchanger<List<Fat>>();
    List<Fat>
      producerList = new CopyOnWriteArrayList<Fat>(),
      consumerList = new CopyOnWriteArrayList<Fat>();
    exec.execute(new ExchangerProducer<Fat>(xc,
      BasicGenerator.create(Fat.class), producerList));
    exec.execute(
      new ExchangerConsumer<Fat>(xc,consumerList));
    TimeUnit.SECONDS.sleep(delay);
    exec.shutdownNow();
  }
}
```

АТОМАРНЫЕ ОПЕРАЦИИ

ЧАСТЬ 4

АТОМАРНАЯ ОПЕРАЦИЯ

- Операция выполняемая как единое целое
  - Чтение
  - Запись
- Неатомарные операции
  - Инкремент
  - Декремент

ВИДЫ АТОМАРНЫХ ОПЕРАЦИЙ

- Чтение
  - get
- Запись
  - set
- Чтение и запись
  - getAndSet
- Условная запись
  - compareAndSet

УСЛОВНАЯ ЗАПИСЬ

- compareAndSet(old, new)
  - Если текущее значение равно old
  - Установить значение в new
- Идиома

```
do {
    old = v.get();
    new = process(old);
} while (v.compareAndSet(old, new));
```

РЕШЕНИЕ ЗАДАЧИ ДОСТУПА К РЕСУРСУ

```
// Получение доступа к ресурсу
while(!v.compareAndSet(0, 1));

// Действия с ресурсом

// Освобождение ресурса
v.set(0);
```

НЕБЛОКИРУЮЩИЙ СЧЕТЧИК

```
public final class Counter {
    private long value = 0;

    public synchronized long
getValue() {
        return value;
    }

    public synchronized long
increment() {
        return ++value;
    }
}
```

```
public class NonblockingCounter {
    private AtomicInteger value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        while (!value.compareAndSet(v,
v + 1));
        return v + 1;
    }
}
```

АТОМАРНЫЕ ОПЕРАЦИИ В JAVA

- Чтение / запись
  - get() – атомарное чтение
  - set(value) – атомарная запись
  - lazySet(value) – запись без барьера
  - getAndSet(value) – чтение и запись
- Проверки
  - compareAndSet(expected, value) – сравнение и запись
  - weakCompareAndSet(expected, value) – слабое сравнение и запись

ОПЕРАЦИИ НАД ЧИСЛАМИ

- Пре- операции
  - getAndIncrement() – инкремент
  - getAndDecrement() – декремент
  - addAndGet() – сложение
- Пост- операции
  - incrementAndGet() – инкремент
  - decrementAndGet() – декремент
  - getAndAdd() – сложение

АТОМАРНЫЕ ПЕРЕМЕННЫЕ

- Типы
  - AtomicBoolean
  - AtomicInteger
  - AtomicLong
  - AtomicReference
- Операции
  - Обычные

АТОМАРНЫЕ МАССИВЫ

- Типы
  - AtomicIntegerArray
  - AtomicLongArray
  - AtomicReferenceArray
- Операции
  - Обычные, с указанием индекса
  - length() – число элементов