Уральский федеральный университет

Параллельные вычисления Многопоточное программирование. Часть 2

Созыкин Андрей Владимирович

к.т.н.
Заведующий кафедрой высокопроизводительных компьютерных технологий
Институт математики и компьютерных наук

Полезные методы потоков

Thread.join():

- Ожидание завершения потока
- Вызов метода блокируется до окончания работы потока

Thread.sleep(long millis):

- Остановка потока на заданное количество милисекунд

Thread.yield():

- Метод сообщает планировщику, что поток выполнил все необходимые действия и не нуждается в процессорном времени
- Процессорное время может быть отдано другому потоку
- Планировщик может игнорировать вызов yield()

Как избежать взаимоблокировки

Захватывать только одну блокировку

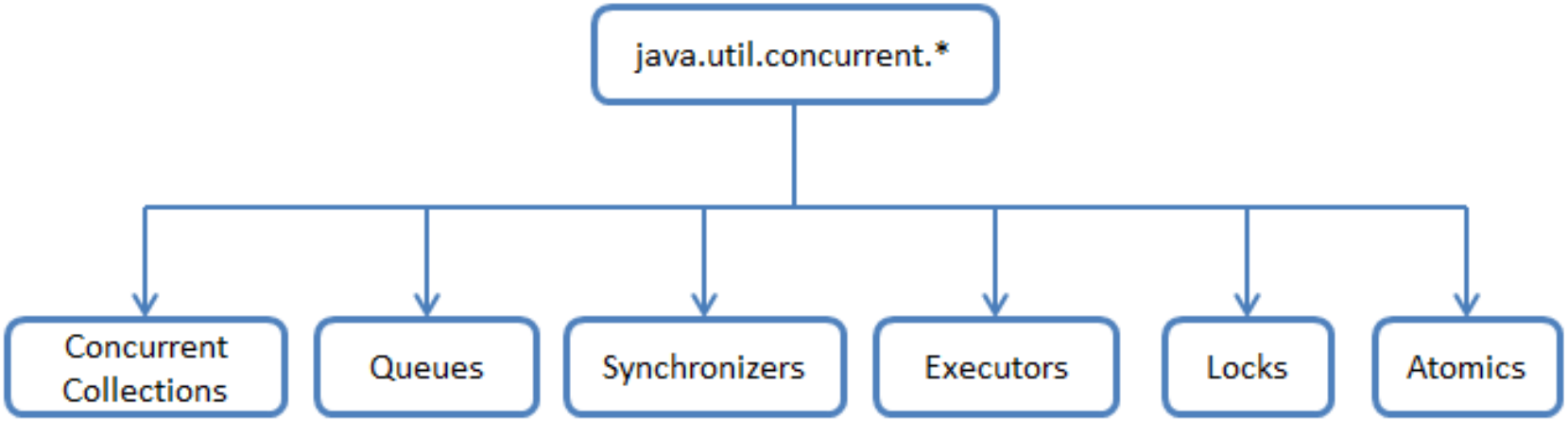
Захватывать блокировки в одном порядке

Добровольно освобождать захваченную блокировку (после таймаута)

Пакет java.util.concurrent

Появился в Java 5

Включает расширенные средства блокировки, управления многопоточностью и готовые многопоточные коллекции



Интерфейс Lock

java.util.concurrent.locks.Lock

```
public interface Lock {
  void lock ();
  void lockInterruptibly () throws InterruptedException ;
  boolean tryLock();
  boolean tryLock( long timeout , TimeUnit unit )
  throws InterruptedException ;
  void unlock();
  ...
}
```

TimeUnit – enum для представления единиц измерения времени:

- TimeUnit.MILLISECONDS, TimeUnit.SECONDS, TimeUnit.MINUTES и т.д.

Как использовать Lock

```
Lock l = new ReentrantLock();
l.lock();
// l.tryLock();
// l.tryLock(1, TimeUnit.SECONDS);
try {
  // Критическая секция
} finally {
  // Обязательно освобождаем блокировку!
  l.unlock();
}
```

ReentrantLock – реализация Lock, которую можно вызывать несколько раз без взаимоблокировки

Банк. Перевод денег с синхронизацией

```
public void transfer(Account from, Account to, int value)
  throws Exception {
    if ( value <= 0) {
      throw new Exception ("Amount must be positive!");
    }
        synchronized ( from ) {
      if (from.getBalance () < value ) {
        throw new Exception ("Not enough money!");
      } else {
        from.post(-value);
      }
      synchronized (to) {
        to.post(value);
      }
    }
}
```

Банк. Перевод денег с синхронизацией v2

```
while (true) {
  if (from.lock.tryLock()) {
    try {
      if (to.lock.tryLock()) {
         try {
           if (from.getBalance () < value ) {
             throw new Exception ("Not enough money!");
           } else {
             from.post(-value);
             to.post(value);
           } finally {
              to.lock.unlock();
           }
       }
    finally {
      from.lock.unlock();
    }
  }
  TimeUnit.NANOSECONDS.sleep(fixedDelay + rnd.nextLong () % randMod);
}
```

Преимущества и недостатки Lock

Больше возможностей по сравнению со стандартными мониторами:

- Асинхронная блокировка (tryLock)
- Задание времени на попытку блокировки
- Блокировка с возможность прерывания

Полностью ручное управление:

- Создавать объект Lock
- Выполнять блокировку
- Не забывать освобождать блокировку, когда она не нужна или когда произошло исключение и т.п.

Атомарные классы

Пакет java.util.concurrent.atomic:

- AtomicInteger
- AtomicLong
- AtomicBoolean

Атомарные операции:

- Выполняются без прерывания
- Используют специальные команды современных процессоров

`AtomicInteger`:

- *`addAndGet`*`(int delta)`
- *`incrementAndGet`*`()`
- *`decrementAndGet`*`()`
- *`getAndSet`*`(int newValue)`

Рекомендации по использованию

1. Мониторы
2. java.util.concurrent.locks
3. Атомарные классы

Базовые сведения об оборудовании

Разработка последовательных программ

- Об оборудовании можно не знать ничего

Разработка параллельных программ:

- Особенности устройства современных многоядерных процессоров существенно влияют на работу многопоточных и параллельных программ

Test and Set Lock

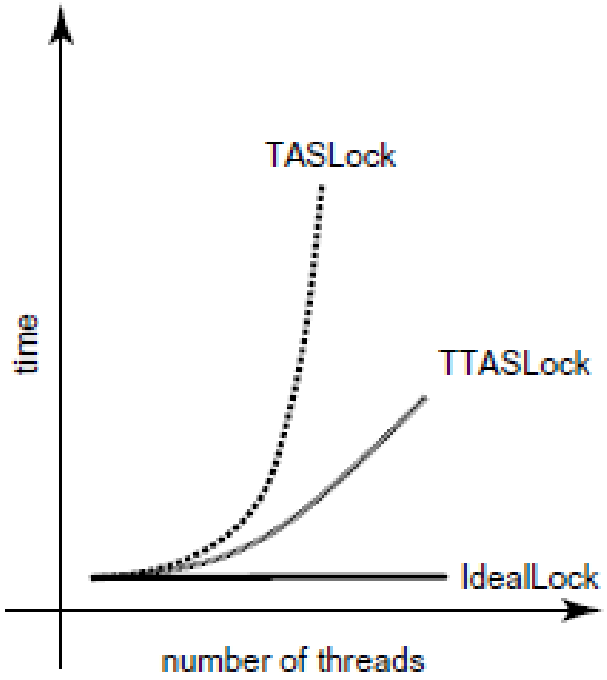
```
public class TASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);
  public void lock() {
    while (state.getAndSet(true)) {}
  }
  public void unlock() {
    state.set(false);
  }
}
```

Test and Test and Set Lock

```
public class TTASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);
  public void lock() {
    while (true) {
      while (state.get()) {};
        if (!state.getAndSet(true))
          return;
    }
  }
  public void unlock() {
    state.set(false);
  }
}
```

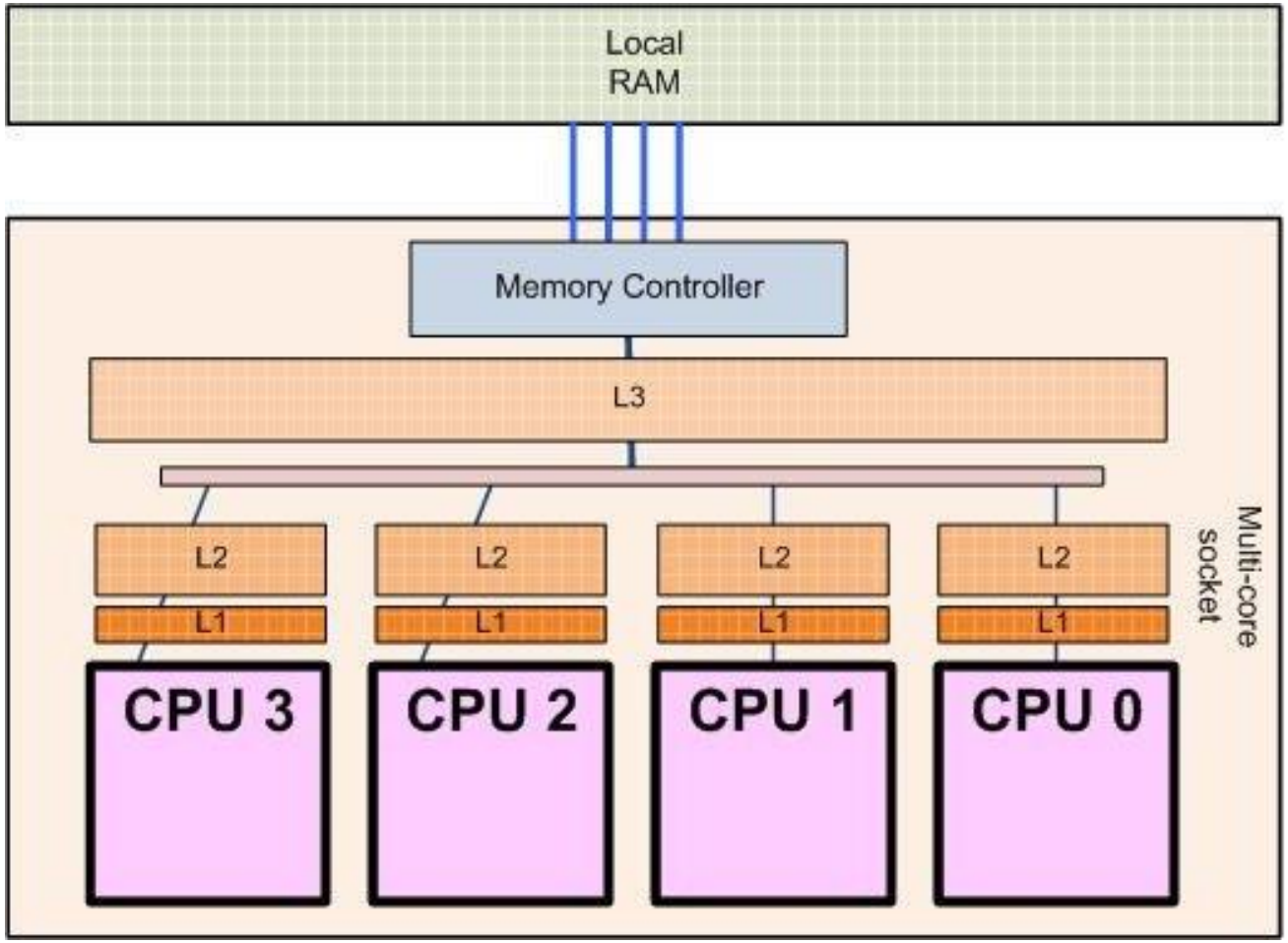
TASLock vs TTASLock

Какая блокировка сработает быстрее?



Maurice Herlihy. Nir Shavit. The Art of Multiprocessor Programming

Иерархия памяти



Ориентировочное время доступа

Кэш 1 уровня ~1-2 такта

Кэш 2 уровня ~10-40 тактов

Кэш 3 уровня ~50-100 тактов

Основная память – сотни тактов

Когерентность кэшей

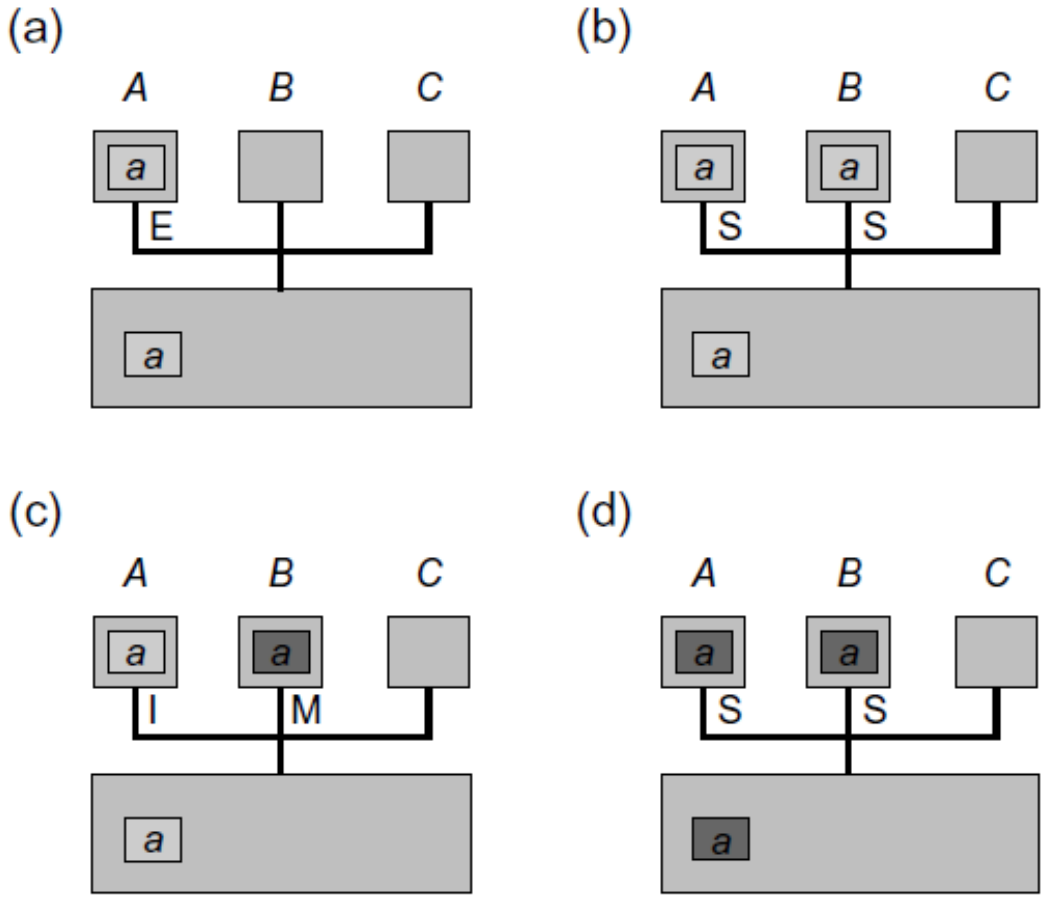
Если один из процессоров (ядер) поменял данные в своем КЭШе, другие процессоры должны узнать об этом

Поддержка когерентности кэшей требует высоких накладных расходов

Протокол MESI:

- **M**odified – данные в кэше были изменены
- **E**xclusive – данные загружены в кэш только одного процессора
- **S**hared – данные загружены в кэш разных процессоров, но не изменены
- **I**nvalid – кэш содержит неправильные данные

Протокол MESI



Maurice Herlihy. Nir Shavit. The Art of Multiprocessor Programming

Когерентность кэшей. Выводы

Требуется разное время на:

- Чтение данных (самое быстрое)
- Запись данных (среднее)
- Уверенность в том, что данные записались в общую память (самое медленное)

Test and Set Lock

```
public class TASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);
  public void lock() {
    // Каждый раз пытаемся ЗАПИСАТЬ данные
    // Высокий трафик для когерентности кэшей
    while (state.getAndSet(true)) {}
  }
  public void unlock() {
    state.set(false);
  }
}
```

Test and Test and Set Lock

```
public class TTASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);
  public void lock() {
    while (true) {
      // Загружаем данные в кэш и читаем из кэша
      // Если состояние поменялось, читаем из общей памяти или
      // кэша другого процессора
      while (state.get()) {};
        // Записываем только когда есть реальная возможность
        if (!state.getAndSet(true))
          return;
    }
  }
  public void unlock() {
    state.set(false);
  }
}
```

Модель памяти

Необходима поддержка иерархии памяти в языке программирования

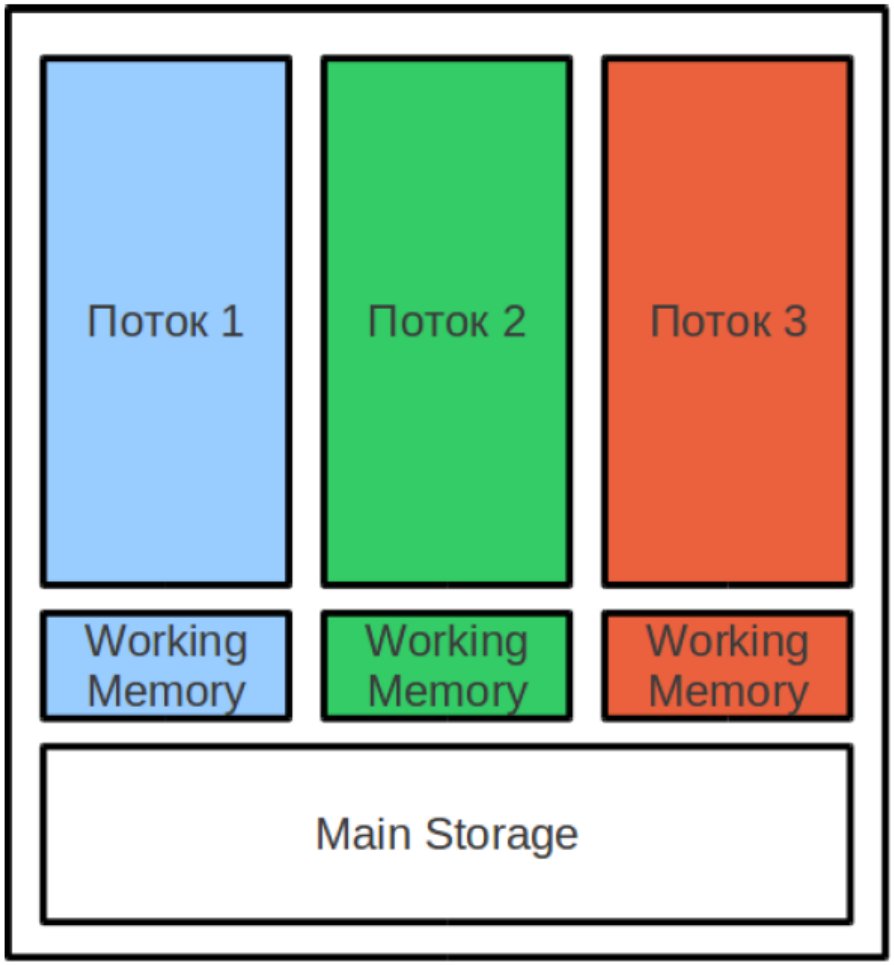
Модель памяти:

- Какие данные являются общими для всех потоков, а какие частными
- Как допустимо переставлять операции доступа в память

Языки программирования:

- Java
- C#
- C++11

Модель памяти Java



Пример использования модели памяти

```
public class TestMemoryModel {
  private static boolean ready;
  private static int number;

  private static class ReaderThread extends Thread {
    public void run () {
      while (!ready)
        Thread.yield();
      System.out.println( number );
    }
  }
  public static void main ( String [] args ) {
    new ReaderThread().start();
    number = 42;
    ready = true ;
  }
}
```

Ключевое слово volatile

Используется для объявления переменных «общими» для всех потоков

Чтение volatile переменной:

- Все копии в кэшах становятся инвалидными
- Данные читаются напрямую из памяти

Запись volatile переменной:

- Данные записываются напрямую в память

Пример использования volatile

```
public class TestMemoryModel {
  private static volatile boolean ready;
  private static int number;

  private static class ReaderThread extends Thread {
    public void run () {
      while (!ready)
        Thread.yield();
      System.out.println( number );
    }
  }
  public static void main ( String [] args ) {
    new ReaderThread().start();
    number = 42;
    ready = true;
  }
}
```

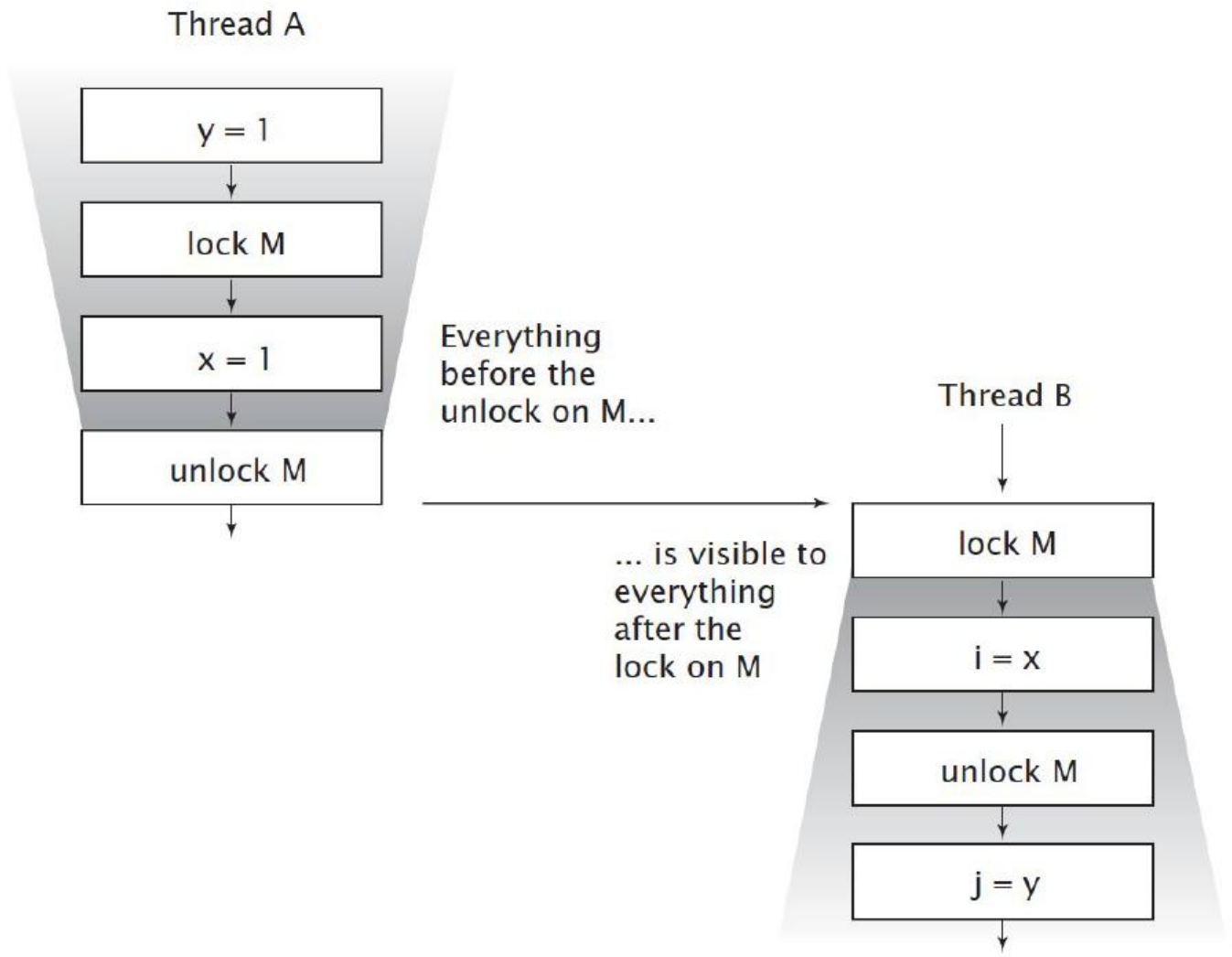
Перестановка операций доступа в память

```
public class TestMemoryModel {
  private static volatile boolean ready;
  private static int number;

  private static class ReaderThread extends Thread {
    public void run () {
      while (!ready)
        Thread.yield();
      System.out.println( number );
    }
  }
  public static void main ( String [] args ) {
    new ReaderThread().start();
    // Меняем местами флаг и установка значения number
    ready = true;
    number = 42;
  }
}
```

Везде volatile!

```
public class TestMemoryModel {
  private static volatile boolean ready;
  private static volatile int number;

  private static class ReaderThread extends Thread {
    public void run () {
      while (!ready)
        Thread.yield();
      System.out.println( number );
    }
  }
  public static void main ( String [] args ) {
    new ReaderThread().start();
    number = 42;
    ready = true;
  }
}
```

Видимость через синхронизацию



Видимость через синхронизацию

```
public class TestMemoryModel {
  private static boolean ready;
  private static volatile int number;
  public static synchronized void stopRequest(){ready = true;}

  private static class ReaderThread extends Thread {
    public void run () {
      while (!ready)
        Thread.yield();
      System.out.println( number );
    }
  }
  public static void main ( String [] args ) {
    new ReaderThread().start();
    number = 42;
    stopRequest();
  }
}
```

volatile vs синхронизация

Синхронизация и volatile позволяют обеспечить «видимость» переменных

Атомарность:

- Гарантирована при синхронизации
- Не гарантирована при volatile

Отсутствие атомарности не всегда плохо:

- Несколько потоков могут дать команду об остановке программы
- Остановка происходит в любом случае, не зависимо от того, какой поток дал команду

Ложное разделение данных

Данные в кэш записываются «строками»

- Размер строки зависит от архитектуры
- Типичный размер 32 или 64 байта

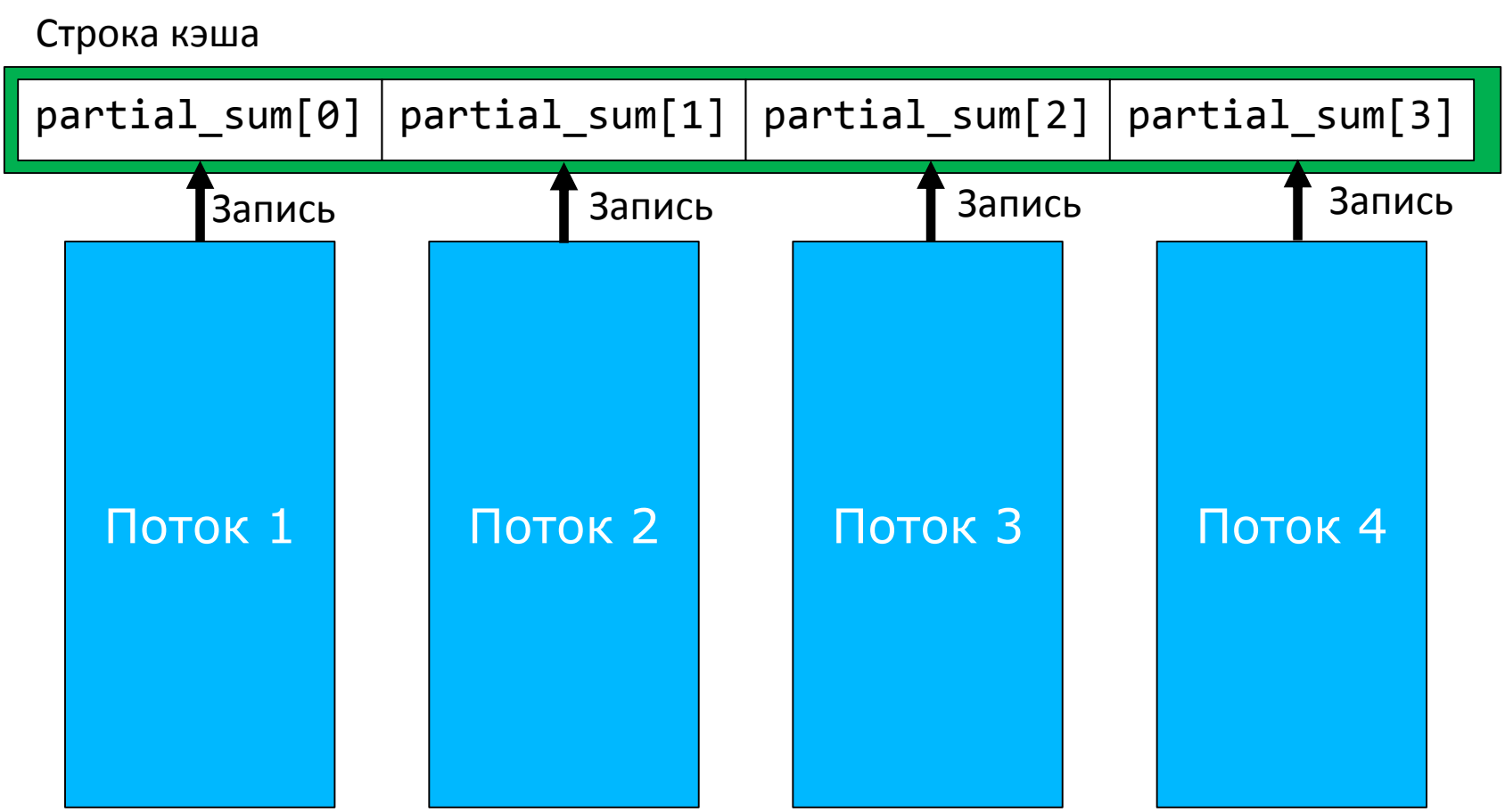
Локальность

- Временная
- Пространственная

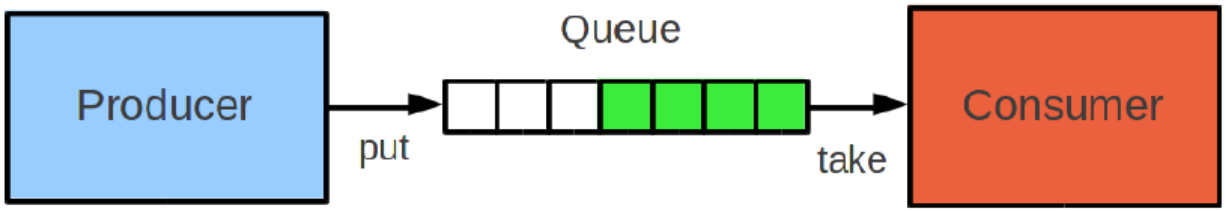
Ложное разделение данных

```
// Общий массив для всех потоков
int[] partial_sum;
partial_sum = new int[threadNum];
…
  // Внутри потока
  public void run () {
    …
    int thread_id = getThreadId();
    for (int i = thread_id ; i < MAX; i += threadNum)
      partial_sum[thread_id] += a[i]; // Ложное разделение
    …
  }
```

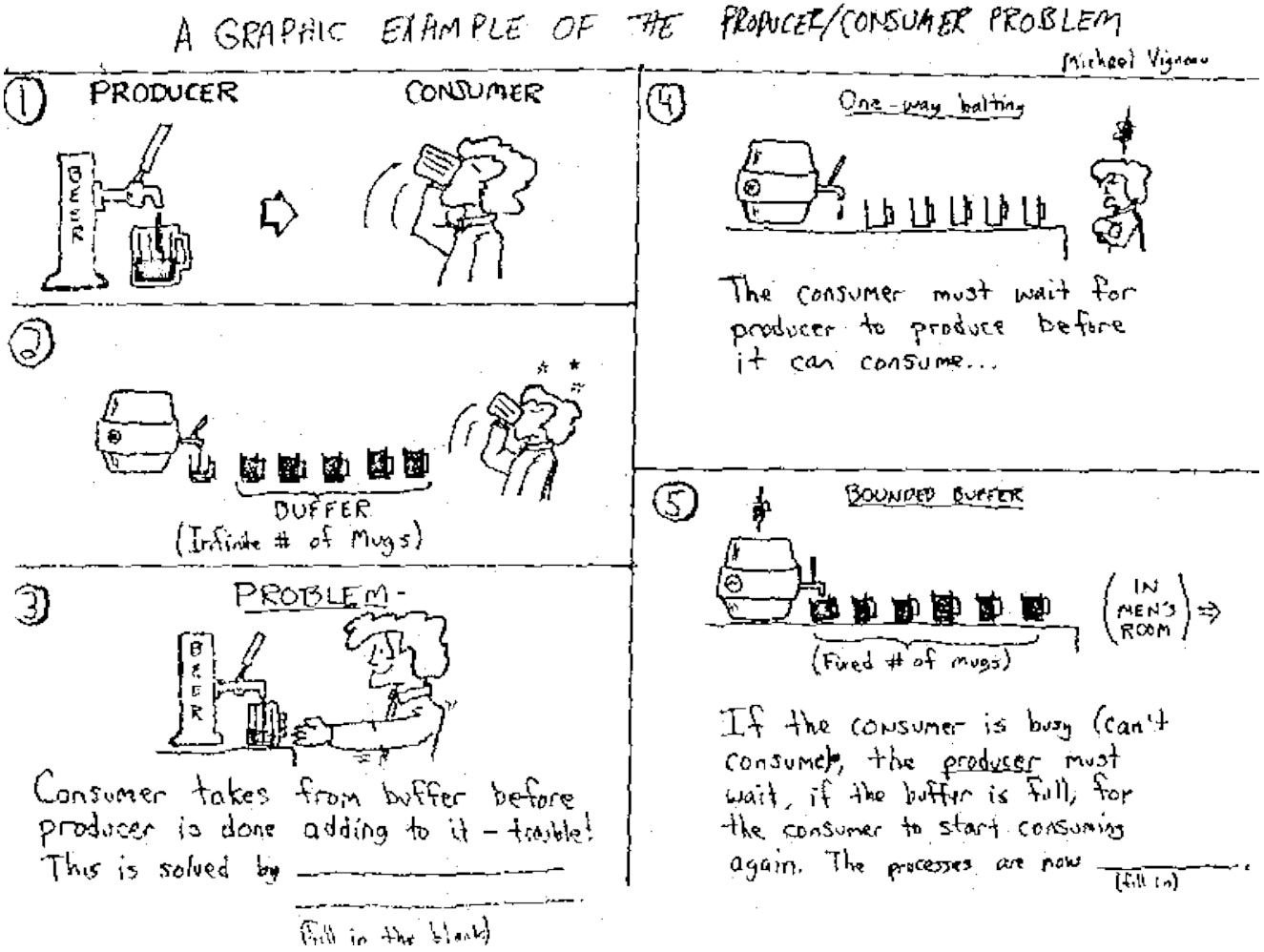
Ложное разделение данных



Проблема производителей и потребителей



Проблема производителей и потребителей



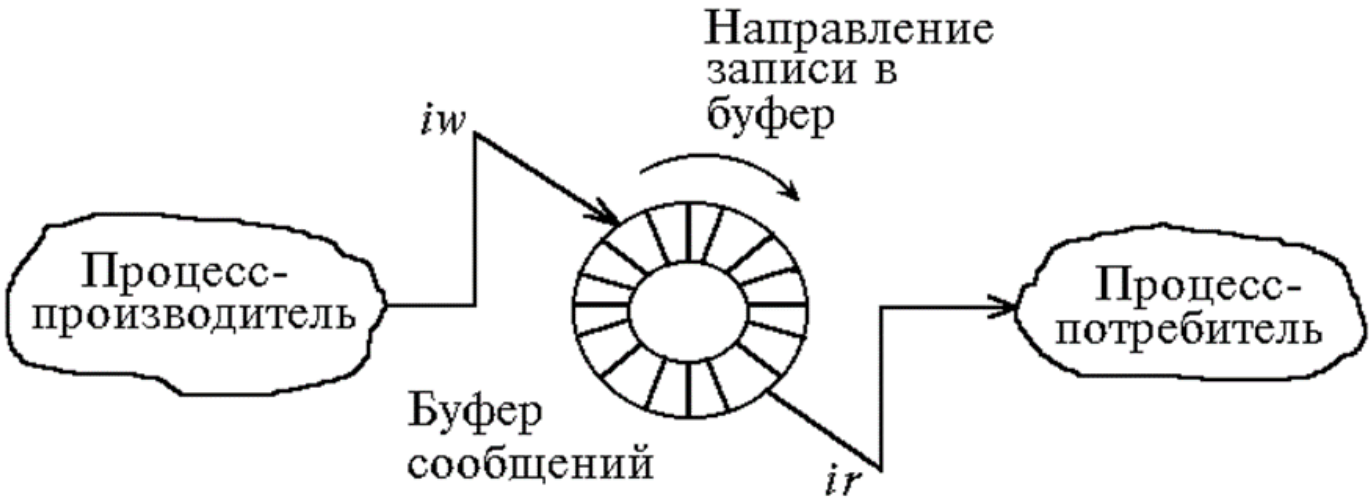
http://www.ccs.neu.edu/home/kenb/synchronize.html

Пример – колл центр

Клиенты, которые звонят в центр – производители

Операторы – потребители

Очередь неотвеченных звонков



Реализация очереди звонков

```
class CallQueue {
  final static int QSIZE = 100; // Размер буфера
  int head = 0; // номер следующего вызова
  int tail = 0; // следующий свободный слот
  Call[] calls = new Call[QSIZE];

  public enq(Call x) { // вызывается АТС
   calls[(tail++) % QSIZE] = x;
  }

  public Call deq() { // вызывается оператором
   return calls[(head++) % QSIZE]
 }
}
```

Очередь звонков с критическими секциями

```
class CallQueue {
  final static int QSIZE = 100; // Размер буфера
  int head = 0; // номер следующего вызова
  int tail = 0; // следующий свободный слот
  Call[] calls = new Call[QSIZE];

  public synchronized enq(Call x) { // вызывается АТС
   calls[(tail++) % QSIZE] = x;
  }

  public synchronized  Call deq() { // вызывается оператором
   return calls[(head++) % QSIZE]
 }
}
```

Очередь звонков с критическими секциями

```
class CallQueue {
  final static int QSIZE = 100; // Размер буфера
  int head = 0; // номер следующего вызова
  int tail = 0; // следующий свободный слот
  Call[] calls = new Call[QSIZE];

  public synchronized enq(Call x) { // вызывается АТС
   calls[(tail++) % QSIZE] = x;
  }

  public synchronized  Call deq() { // вызывается оператором
   return calls[(head++) % QSIZE]
 }
}
```

Что будет, если оператор извлекает звонок, а очередь пуста?

Пустая очередь звонков

```
public synchronized T deq() {
  while (head == tail) {}; // Ожидаем, пока очередь пуста
  return call[(head++) % QSIZE];
}
```

Пустая очередь звонков

```
public synchronized T deq() {
  while (head == tail) {}; // Ожидаем, пока очередь пуста
  return call[(head++) % QSIZE];
}
```

Используется активное ожидание:

- Постоянно проверяется выполнение условия
- Высокая «ненужная» нагрузка на процессор

Пустая очередь звонков

```
public synchronized T deq() {
  while (head == tail) {}; // Ожидаем, пока очередь пуста
  return call[(head++) % QSIZE];
}
```

Используется активное ожидание:

- Постоянно проверяется выполнение условия
- Высокая «ненужная» нагрузка на процессор

Взаимоблокировка:

- В синхронизованном методе deq ждем, когда кто-то выполнит синхронизованный метод enq

Пустая очередь звонков

```
public synchronized T deq() {
  while (head == tail) {
    wait()
  };
  return call[(head++) % QSIZE];
}
```

Метод wait():

- Составная часть монитора, есть у каждого объекта
- Освобождает блокировку
- Останавливает поток
- Завершается после вызова метода notifyAll() того же объекта

Пустая очередь звонков

```
public synchronized T deq() {
  while (head == tail) {
    wait()
  };
  return call[(head++) % QSIZE];
}

public synchronized enq(Call x) {
  calls[(tail++) % QSIZE] = x;
  if (tail-head == QSIZE-1) {
    notifyAll();
  }
}
```

notifyAll() и notify()

Метод notifyAll():

- Будит **все потоки**, которые вызвали метод wait() на данном объекте
- Один из потоков входит в критическую секцию, остальные ждут

Метод notify():

- Будит **один из потоков**, которые вызвали метод wait() на данном объекте
- Какой именно поток будет разбужен предсказать нельзя
- Не рекомендуется использовать из-за проблемы «lost wakeup»

Полная очередь звонков

```
public synchronized enq(Call x) {
  while (tail - head == QSIZE) { wait()};
  calls[(tail++) % QSIZE] = x;
  if (tail-head == QSIZE-1) {
    notifyAll();
  }
}
public synchronized T deq() {
  while (head == tail) { wait()};
  Call temp = call[(head++) % QSIZE];
  if (tail-head == 1) {
    notifyAll();
  }
  return temp;
}
```

Проблемы notifyAll()

notifyAll() будит все потоки:

- Потоки, которые ждут вставки звонка в полную очередь
- Потоки, которые ждут извлечения звонка из пустой очереди
- Нужно будить только потоки из первой или второй категории

Решение `java.util.concurrent.locks.Condition`:

- Позволяет останавливать потоки выборочно
- `condition.await()`
- `condition.await(long time, TimeUnit unit)`
- `condition.signallAll()`

Очередь звонков с Condition

```
class CallQueue {
  final static int QSIZE = 100; // Размер буфера
  int head = 0; // номер следующего вызова
  int tail = 0; // следующий свободный слот
  Call[] calls = new Call[QSIZE];

  protected final Lock lock = new ReentrantLock();
  private final Condition notFull = lock.newCondition();
  private final Condition notEmpty = lock.newCondition();
  …
}
```

Очередь звонков с Condition

```
public synchronized enq(Call x) {
  while (tail - head == QSIZE) { notFull.await()};
  calls[(tail++) % QSIZE] = x;
  if (tail-head == 1) {
    notEmpty.notifyAll();
  }
}
public synchronized T deq() {
  while (head == tail) { notEmpty.await()};
  Call temp = call[(head++) % QSIZE];
  if (tail-head == QSIZE - 1) {
    notFull.notifyAll();
  }
  return temp;
}
```

Готовые реализации очередей

Интерфейс `java.util.concurrent.BlockingQueue`:

- Многопоточная (thread safe) реализация очереди
- Блокировка при записи в полную очередь
- Блокировка при извлечнии из пустой очереди

Реализации:

- `ArrayBlockingQueue`
- `LinkedBlockingQueue`
- `PriorityBlockingQueue`
- `DelayQueue`
- `SynchronousQueue`

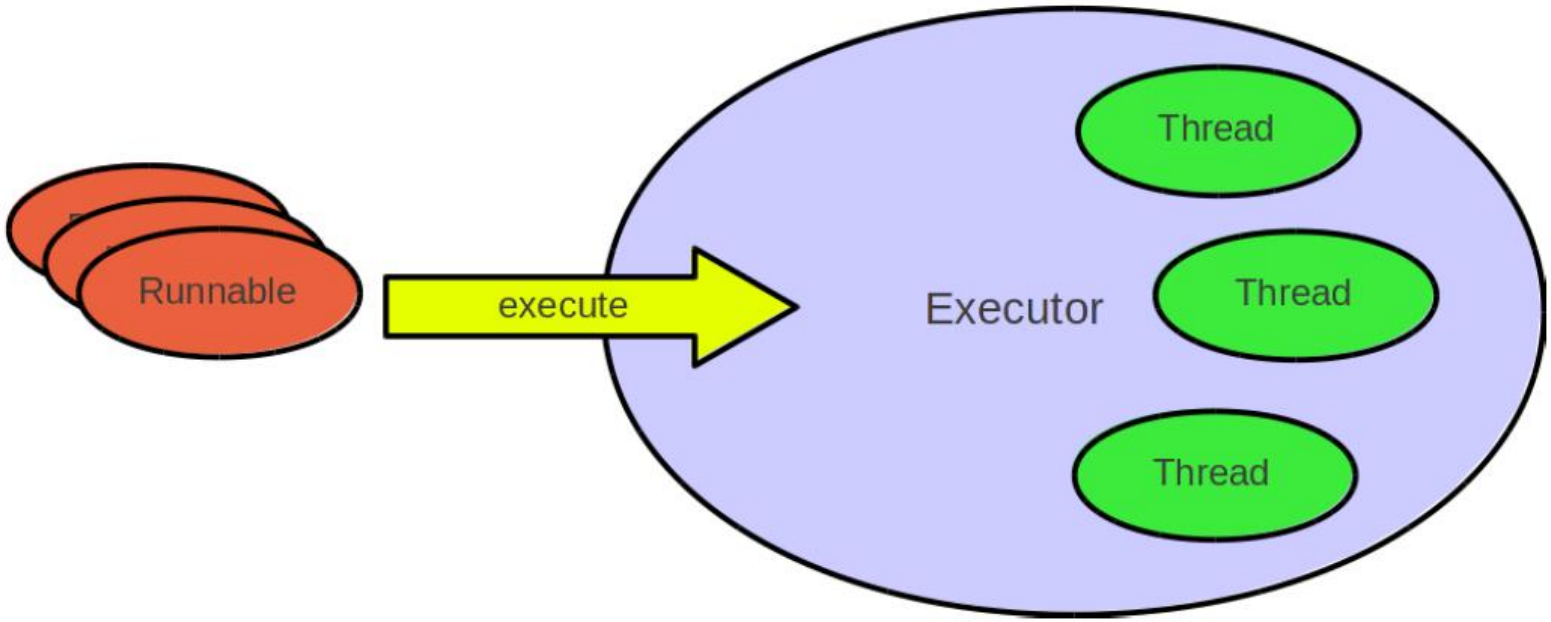
Управление потоками в Java

Сколько потоков нужно вашей программе?

Когда и как создавать эти потоки?

Какой поток какую задачу будет решать?

Executor



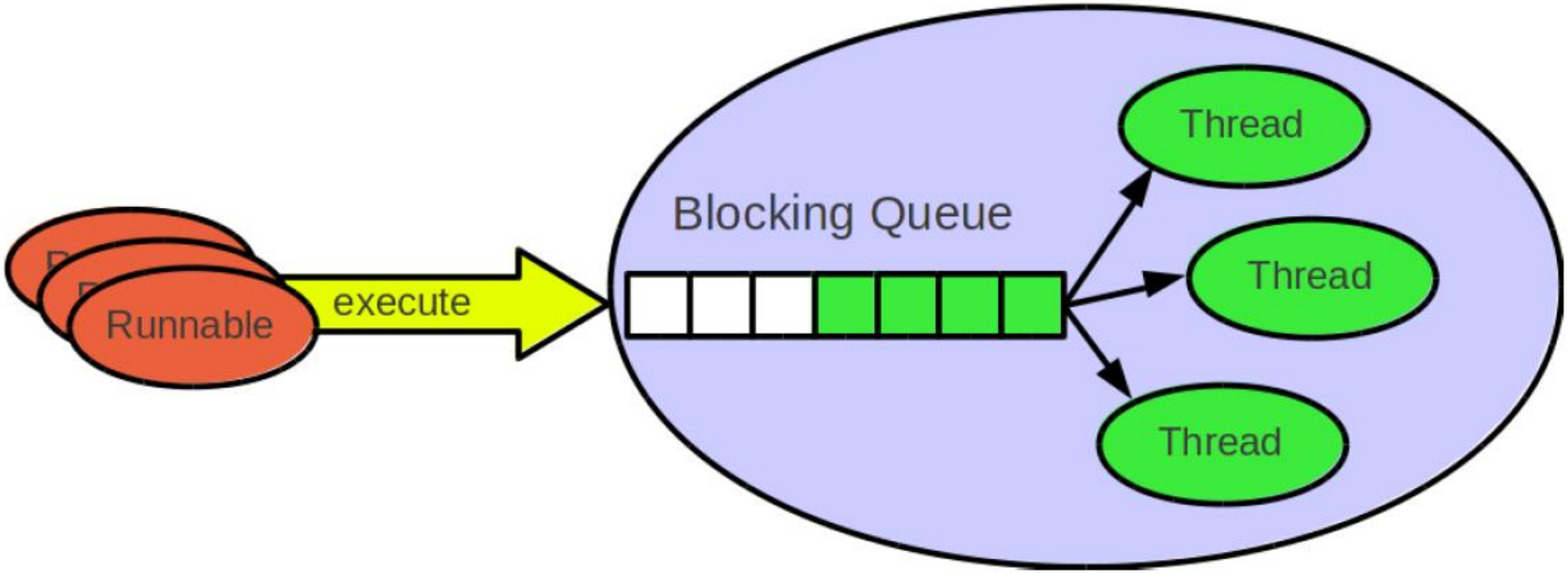
Класс для потока

```
public class HelloWorld implements Runnable {
  private int id;
  public HelloWorld(int id){
    this.id = id;
  }
  public void run() {
    for (int i = 0; i < 10; i++)
      System.out.println("Hello from thread # " + id);
  }
}
```

Запуск с помощью Executor

```
public class CachedThreadPool {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new HelloWorld(i));
    exec.shutdown();
  }
}
```

Executor




```
ThreadPoolExecutor (int corePoolSize ,
  int maximumPoolSize,
  long keepAliveTime,
  TimeUnit unit,
  BlockingQueue<Runnable>workQueue)
```

Готовые Executor’ы

```
ExecutorService.newCachedThreadPool()
   .newFixedThreadPool(int nThreads)
   .newSingleThreadExecutor()
   .newScheduledThreadPool(int corePoolSize)

// Реализации в OpenJDK
public static ExecutorService newFixedThreadPool(int nThreads) {
  return new ThreadPoolExecutor(nThreads, nThreads,
    0L, TimeUnit.MILLISECONDS,
    new LinkedBlockingQueue<Runnable>());
 }

public static ExecutorService newCachedThreadPool() {
  return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
     60L, TimeUnit.SECONDS,
     new SynchronousQueue<Runnable>());
}
```

Как вернуть значение из потока?

Интерфейс `Runnable`:

- `void run()`

Интерфейс `Callable<T>`:

- `T call()`
- Возвращает значение типа T

Запуск потока `Callable<T>`:

- `Future<T> ExecutorService.submit(Callable<T> task)`

Интерфейс `Future<T>`:

- `boolean isDone()`
- `T get() / get(long timeout, TimeUnit unit)`
- `boolean cancel(boolean mayInterruptIfRunning)`

Числа Фибоначчи

```
class FibTask implements Callable<Integer> {
  static ExecutorService exec = Executors.newCachedThreadPool();
  int arg;
  public FibTask(int n) {
    arg = n;
  }
  public Integer call() {
    if (arg > 2) {
      Future<Integer> left = exec.submit(new FibTask(arg-1));
      Future<Integer> right = exec.submit(new FibTask(arg-2));
      return left.get() + right.get();
    } else {
      return 1;
    }
  }
}
```

Вопросы?