СБЕРБАНК ТЕХНОЛОГИИ

MULTITHREADING part 2

УЗНАЕМ

СБЕРБАНК ТЕХНОЛОГИИ

- Что такое общие ресурсы? Зачем они нужны?
- Основные примитивы синхронизации.
- Как останавливать работу потока.
- Кооперация между потоками.
- Основные проблемы многопоточного кода

ОБЩИЕ РЕСУРСЫ

СБЕРБАНК ТЕХНОЛОГИИ

В многопоточном коде в отличие от однопоточного возникает проблема, когда два и более потоков пытаются использовать один и тот же ресурс.

Может возникнуть не консистентное состояние общего объекта.

ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Определим интерфейс баланса:

```
class Account {
    private int balance = 50;
    public int getBalance() {
        return balance;
    }
    public void withdraw(int amount) {
        balance = balance - amount;
    }
}
```

Зададим функцию для снятия денег:

```
public class AccountExample {
    private Account acct = new Account();

    private void makeWithdrawal(int amt) {
        if (acct.getBalance() >= amt) {
            acct.withdraw(amt);
        }
    }
```

ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Снимать будем по 10 единиц, 5 раз, пока баланс не иссякнет.

```
public void run() {
    for (int x = 0; x < 5; x++) {
        makeWithdrawal(10);
        if (acct.getBalance() < 0) {
            System.out.println("account is overdrawn!");
        }
    }
}
```

Какой будет вывод у программы, если запустить в 2 потока?

ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

```
account is overdrawn!
Not enough in account for Thread-1 to withdraw -10
account is overdrawn!
Not enough in account for Thread-0 to withdraw -10
account is overdrawn!

Process finished with exit code 0
```

Это проблема известна как «гонка за ресурсами» когда, несколько потоков пытаются получить доступ к одному и тому же ресурсу и приводит к «порче» ресурса.

Всё что нужно гарантировать чтобы единовременно только один поток мог выполнять код функции снятия денег makeWithdrawal.

Т.е. должны гарантировать атомарность операции.

Что нужно сделать в java для достижения целей атомарности:

1. Применить модификатор доступа **private** для общих полей
2. Синхронизировать доступ к общим полям с помощью ключевого слова **synchronized**

РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Что нужно сделать в java для достижения целей атомарности:

1. Применить модификатор доступа **private** для общих полей
2. Синхронизировать доступ к общим полям с помощью ключевого слова **synchronized**

```
private synchronized void makeWithdrawal(int amt) {
    // without changes
}
```

Вывод предыдущего примера:

```
Not enough in account for Thread-0 to withdraw 0
Not enough in account for Thread-1 to withdraw 0
Not enough in account for Thread-1 to withdraw 0
Not enough in account for Thread-1 to withdraw 0
Not enough in account for Thread-1 to withdraw 0
```

JAVA OBJECT И MONITOR

СБЕРБАНК ТЕХНОЛОГИИ

Каждый объект в java имеет встроенный монитор и он работает как мьютекс.

```
public synchronized void invoke() {
    //do some logic thread safely
}
```

JAVA OBJECT И MONITOR

СБЕРБАНК ТЕХНОЛОГИИ

Каждый класс в java имеет встроенный монитор и он работает как мьютекс.

```
public static synchronized void invoke() {
    //do some logic thread safely
}
```

ЗАЩИТА ОБЩИХ РЕСУРСОВ С SYNCHRONIZED

СБЕРБАНК ТЕХНОЛОГИИ

К общим ресурсами могут являться и область памяти, файлы, I/O порты и т.п.

КРИТИЧЕСКАЯ СЕКЦИЯ

СБЕРБАНК ТЕХНОЛОГИИ

Так как синхронизация уменьшает выигрыш по производительности от параллельной обработки, следовательно нужно уменьшать её скоуп и синхронизировать не больше того кода который необходим для защиты общих данных.

Такая область называется критической секцией.

Нужен механизм синхронизировать доступ не ко всей функции, а только к части.

КРИТИЧЕСКАЯ СЕКЦИЯ

СБЕРБАНК ТЕХНОЛОГИИ

В java критическую секцию можно задать так:

```
synchronized (lock_object) {
    // accessed only one task at a time
}
```

КРИТИЧЕСКАЯ СЕКЦИЯ - ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

```
public void increment() {
    Pair temp;
    synchronized (this) {
        p.incrementX();
        p.incrementY();
        temp = getPair();
    }
    storeToDataBase(temp); // long operation
}
```

Следуя принципу инкапсуляции, лучше было бы спрятать механизм синхронизации в сам класс.

```
public class PrivateLockExample {
    private final Object myLock = new Object();

    void someMethod() {
        synchronized(myLock) {
            // Access or modify the shared data
        }
    }
}
```

Таким образом:

- Мы запрещаем пользователю участвовать в нашей политики синхронизации корректно или нет
- Уменьшаем скоуп поиска потенциальных concurrent проблем
- Теперь можно использовать несколько мониторов для ортогональных данных

JAVA MONITOR PATTERN – ОРТОГОНАЛЬНЫЕ ДАННЫЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
public class Cube {
    private final Object volumeLock = new Object();
    private final Object positionLock = new Object();
    private int length, width, height;
    private int x, y, z;
    void increaseVolume() {
        synchronized(volumeLock) {
            ++length; ++width; ++height;
        }
    }
    void move() {
        synchronized(positionLock) {
            ++x; ++y; ++z;
        }
    }
}
```

ЗАЩИТА ОБЩИХ РЕСУРСОВ С SYNCHRONIZED

СБЕРБАНК ТЕХНОЛОГИИ

Важно:

- К общим ресурсами могут являться и область памяти, файлы, I/O порты и т.п.
- Любой объект содержит в себе один встроенный монитор
- Любой класс содержит в себе один встроенный монитор

ЗАЩИТА ОБЩИХ РЕСУРСОВ С SYNCHRONIZED

СБЕРБАНК ТЕХНОЛОГИИ

Важно:

- В рамках одной задачи можно захватывать монитор больше одного раза из одного потока
- Каждый метод, который имеет доступ к общему ресурсу должен быть синхронизирован
- Единовременное выполнение блока кода под synchronized из разных потоков НЕ возможно

ЗАЩИТА ОБЩИХ РЕСУРСОВ С SYNCHRONIZED

СБЕРБАНК ТЕХНОЛОГИИ

Важно:

- Если ресурс занят то поток встаёт в ожидании на входе в метод до того пока захвативший монитор поток не отпустит его
- Один поток может одновременно захватить несколько мониторов
- Необходимо уменьшать область кода под монитором
- Monitor pattern является более предпочтительным

СИНХРОНИЗИРОВАННЫЕ КОЛЛЕКЦИИ

СБЕРБАНК ТЕХНОЛОГИИ

Можно получить потокобезопасную обёртку любой коллекции:

```
Collection<String> names =
      Collections.synchronizedCollection(new LinkedList<>());

List<String> names =
      Collections.synchronizedList(new LinkedList<>());

Set<String> names =
      Collections.synchronizedSet(new HashSet<>());

Map<String, String> fullName =
      Collections.synchronizedMap(new HashMap<String, String>);
```

СИНХРОНИЗИРОВАННЫЕ КОЛЛЕКЦИИ – ПРЕДОСТЕРЕЖЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
public class SyncCollectionsWarning {
    private List<String> names =
       Collections.synchronizedList(new LinkedList<>());
    // thread safe
    public void add(String name) {
        names.add(name);
    }
    //not thread save
    public String removeFirst() {
        if(names.size() > 0) {
            return names.remove(0);
        }
        return null;
    }
}
```

ЗАДАЧА

СБЕРБАНК ТЕХНОЛОГИИ

Какие проблемы в следующем коде?

```
public class Test implements Runnable {
    private boolean endFlag = false;
    public void end() {
        endFlag = true;
    }
    public void run() {
        while(!endFlag) {
            //do some tasks
        }
    }
}
```

Один поток не обязан видеть изменения другого. И может их не увидеть никогда.

Синхронизация решает этот вопрос.

Так же можно использовать ключевое слово языка volatile:

```
public class Test implements Runnable {
    private volatile boolean endFlag = false;
    public void end() {
        endFlag = true;
    }
    public void run() {
        while(!endFlag) {
            //do some tasks
        }
    }
}
```

Что даёт volatile:

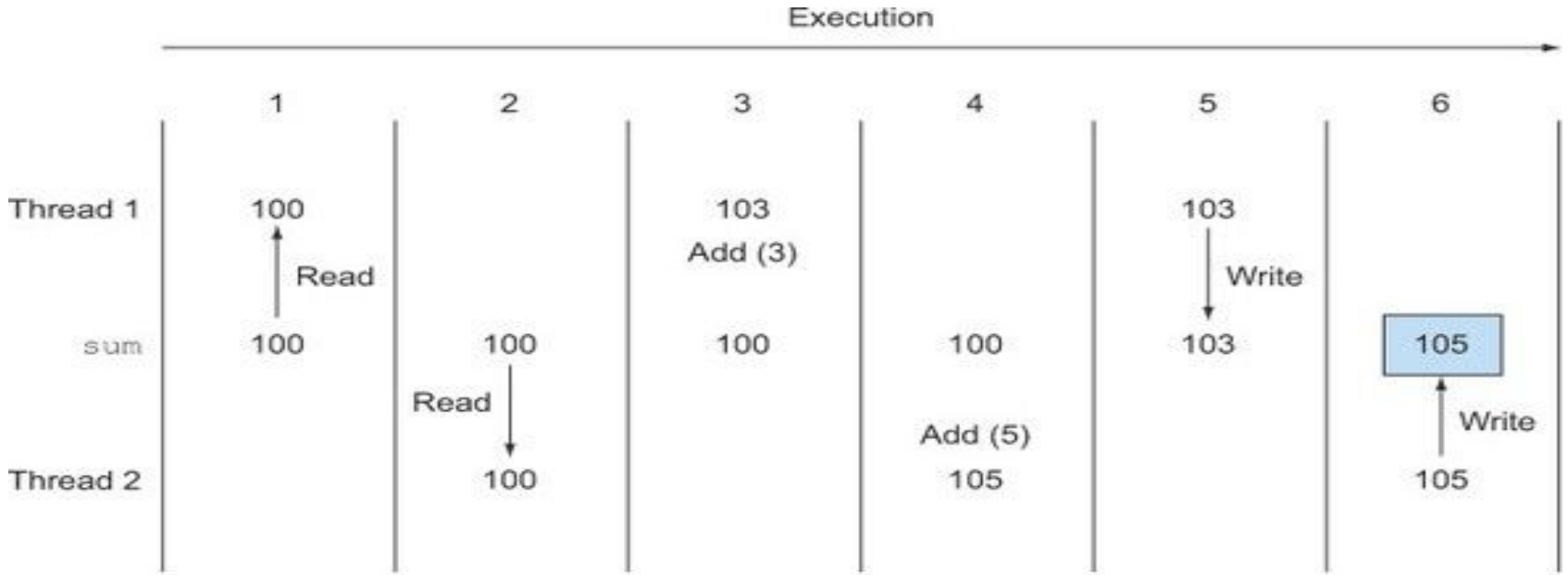
- Поток при чтении обязательно увидит самые актуальные изменения
- На 32 битных платформах позволяет атомарно считывать и писать double и long переменные (не путать с атомарными операциями!)

Ограничения:

- Не может участвовать в инварианте с другими state переменными
- Не работает, если завязаться на предыдущее состояние перед обновлением – не АТОМАРНО

СБЕРБАНК ТЕХНОЛОГИИ

- Обычный инкремент – это 3 действия!



НЕИЗМЕНЯЕМЫЕ ОБЪЕКТЫ

СБЕРБАНК ТЕХНОЛОГИИ

Immutable объекты – такие объекты, которые обладают следующими свойствами:

- Состояние объекта НЕ может изменяться после его создания
- Все его поля финальные final
- Ссылка на объект как this ни куда не передавалась из конструктора

ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Является ли следующий объект неизменяемым?

```
public class Man {
    private final String name;
    private final Date date;
    public Man(String name, Date date) {
        this.name = name;
        this.date = date;
    }
    public String getName() {
        return name;
    }
    public Date getDate() {
        return date;
    }
}
```

Нет – нарушено правило 1:

```
public static void main(String[] args) {
    Man andrey = new Man("Andrey", new Date());

    andrey.getDate().setTime(0);
}
```

Нет – нарушено правило 1:

```
public static void main(String[] args) {
    Man andrey = new Man("Andrey", new Date());

    andrey.getDate().setTime(0);
}
```

Или даже так:

```
public static void main(String[] args) {
    Date d = new Date();
    Man andrey = new Man("Andrey", d);
    d.setTime(0);
}
```

ПРАВИЛЬНЫЙ ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

```
public class Man {
    private final String name;
    private final Date date;
    public Man(String name, Date date) {
        this.name = name;
        this.date = new Date(date.getTime());
    }
    public String getName() {
        return name;
    }
    public Date getDate() {
        return new Date(date.getTime());
    }
```

НЕИЗМЕНЯЕМЫЕ ОБЪЕКТЫ - ОСОБЕННОСТЬ

СБЕРБАНК ТЕХНОЛОГИИ

Immutable объекты - могут использоваться безопасно из любого потока без дополнительной синхронизации.

Если они правильно были опубликованы:

- Статическая инициализация или
- Сохранение как volatile поле или
- Сохранение как final поле

IMMUTABLE ОБЪЕКТЫ В JDK

СБЕРБАНК ТЕХНОЛОГИИ

- BigInteger – всегда возвращает защищённую копию самого себя
- Все классы Date-Time Package для java 8 – так возвращают защищённую копию самого себя
- Группа функций типа Collections.unmodifiable… которые возвращают неизменяемое представление коллекций

ОСТАНОВКА ПОТОКА

СБЕРБАНК ТЕХНОЛОГИИ

Иногда нужно остановить один поток из другого не дождавшись его завершения.

Возможные причины:

- Пользовательский запрос (нажал кнопку «cancel»)
- Таймаут на операцию
- Изменились входные условия – задача больше актуальна
- Выключение сервиса

ОСТАНОВКА ПОТОКА

СБЕРБАНК ТЕХНОЛОГИИ

В java нет безопасного способа остановить поток.

Существует механизм когда один поток запрашивает завершение другого, а другой как-то обрабатывает это условие и по возможности как можно скорее завершается.

ОСТАНОВКА ПОТОКА – СВОЁ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Зная о volatile попробуем решить задачу.

Вспомним уже знакомы код:

```
public class Test implements Runnable{
    private volatile boolean endFlag = false;
    public void end() {
        endFlag = true;
    }
    public void run() {
        while(!endFlag) {
            //do some tasks
        }
    }
}
```

Вроде проблем нет – должно работать.

ОСТАНОВКА ПОТОКА – СВОЁ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Теперь немного изменим код:

```
    public void run() {
        while(!endFlag) {
            //do some tasks
            try {
                //sleep after work
                TimeUnit.MINUTES.sleep(10);
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}
```

ОСТАНОВКА ПОТОКА – СВОЁ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Теперь немного изменим код:

```
public void run() {
    while(!endFlag) {
        //do some tasks
        try {
            //sleep after work
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            return;
        }
    }
}
```

Проблема – поток заснул на 10 минут (blocked state)

хотя мы его прервали!

BLOCKED STATE

СБЕРБАНК ТЕХНОЛОГИИ

Вспомним про состояния потоков.

Thread.sleep переводит нас в blocked и мы не планируемся и ни как не можем обработать завершение.

Так же следующие операции переводят поток в это состояние:

- Object.wait()
- I/O операции, например блокирующее чтение с сокета
- Попытка захватить монитор, когда он уже захвачен другим потоком synchronized

ОСТАНОВКА ПОТОКА – ПРАВИЛЬНОЕ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

В классе Thread есть специальные методы для управления прерыванием потока:

```
    public class Thread {
        public void interrupt() { ... }
        public boolean isInterrupted() { ... }
        public static boolean interrupted() { ... }
...
    }
```

ОСТАНОВКА ПОТОКА – ПРАВИЛЬНОЕ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Перепишем пример:

```
public void run() {
    while(!Thread.currentThread().isInterrupted()) {
        //do some tasks
        try {
            //sleep after work
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

ОСТАНОВКА ПОТОКА – ПРАВИЛЬНОЕ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Перепишем пример:

```
    public static void main(String[] args)
                        throws InterruptedException {
        Thread t = new Thread(new Test2());
        t.start();
        t.interrupt();
        t.join();
    }
}
```

THREAD INTERRUPT

СБЕРБАНК ТЕХНОЛОГИИ

Важно:

- Interrupt() – ни к чему не обязывает – является просто запросом на завершение – выставляет внутренний флаг
- isInterrupted() – позволяет считывать запрос на завершение
- Блокирующие операции выкидывают исключение InterruptedException, но не все
- Статус прерывания сбрасывается в исключении
- Ни как не влияет на попытку захватить уже захваченный другим потоком монитор (synchronized)
- К сожалению не влияет на блокирующие I/O операции

Решение

- Вместо synchronized можно использовать java.util.concurrent.locks.ReentrantLock
- Классические I/O операции можно прервать параллельно с interrupt() вызвав close()
- Вместо классических I/O операций можно использовать классы из nio пакета

ВЗАИМОДЕЙСТВИЕ МЕЖДУ ПОТОКАМИ

СБЕРБАНК ТЕХНОЛОГИИ

Что если нужно дождаться сигнала из другого потока?

Или послать сигнал в другой поток или потоки?

Java имеет встроенный механизм позволяющий потокам взаимодействовать друг с другом.

Каждый объект имеет методы wait(), notify(), notifyAll().

СБЕРБАНК ТЕХНОЛОГИИ

Определим получателя:

```
public void doMessages() {
    try {
        synchronized (this) {
            while (message == null) {
                wait();
            }
            System.out.println("Recv: " + message);
        }
    } catch (InterruptedException e) {
        return;
    }
}
```

Отправим сообщение получателю:

```
public void sendMessage(String message) {
    synchronized (this) {
        this.message = message;
        notify();
    }
}
```

ВАЖНО - WAIT

СБЕРБАНК ТЕХНОЛОГИИ

- Wait позволяет избежать busy waiting для интересуемого события и прерывает поток
- Wait обязан быть вызван в synchronized блоке, иначе IllegalMonitorStateException
- Wait и synchronized должны быть вызваны на одном и том же объекте

ВАЖНО - WAIT

СБЕРБАНК ТЕХНОЛОГИИ

- Wait блокирует поток до тех пор пока из другого не вызовут notify или notifyAll
- Wait вызванный в синхронном методе внутри себя отпускает блокировку
- Wait без параметра ожидает вечно
- Wait с параметром ожидает заданное кол-во миллисекунд

ВАЖНО - WAIT

СБЕРБАНК ТЕХНОЛОГИИ

- Wait может быть прерван, когда прерывается поток – выкидывается InterruptedException
- Wait на некоторых платформах может быть прерван по причине Spurious Wake-Up
- Wait должен работать вместе с проверкой интересуемого условия в цикле
- Проверка условия должна быть под монитором во избежание проблемы потерянного сигнала

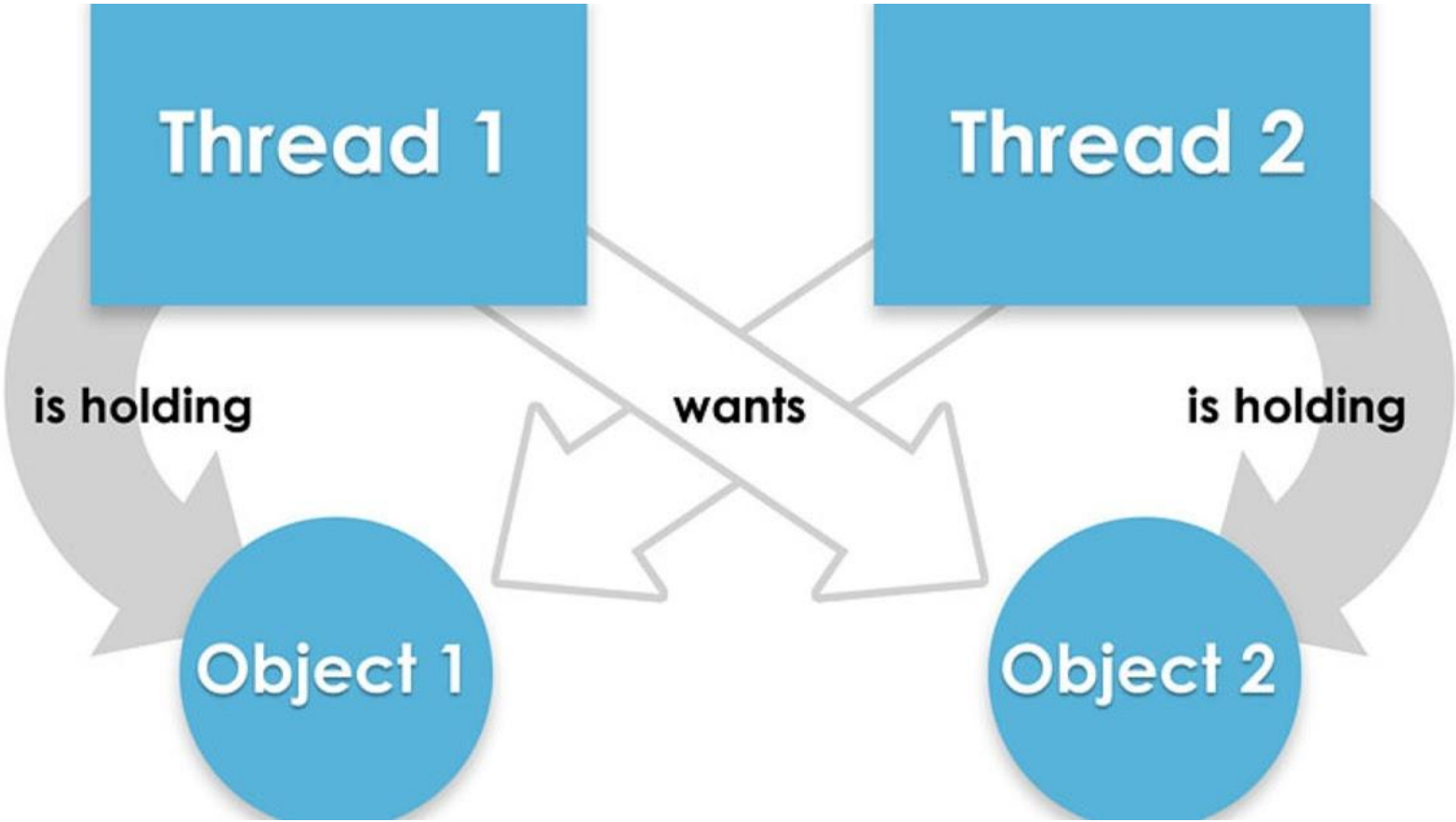
ВАЖНО - NOTIFY AND NOTIFYALL

СБЕРБАНК ТЕХНОЛОГИИ

- Notify и notifyAll позволяют послать сигнал ожидающим на том же объекте потокам
- Notify посылает только один сигнал, если ожидающих потоков несколько только один получит его
- Notify и notifyAll обязаны быть вызваны в synchronized блоке, иначе IllegalMonitorStateException
- Notify и synchronized должны быть вызваны на одном и том же объекте
- Notify и notifyAll предварительно проставляют посылаемое событие

ПРОБЛЕМЫ КОНКУРЕНТНОГО КОДА - DEADLOCK

СБЕРБАНК ТЕХНОЛОГИИ



ПРОБЛЕМЫ КОНКУРЕНТНОГО КОДА - STARVATION

СБЕРБАНК ТЕХНОЛОГИИ

Голод потоков – ситуация, когда один поток регулярно не может получить доступ к общему ресурсу и не может прогрессировать дальше из-за этого.

Возникает когда жадный поток очень часто и на долго захватывает доступ к общему ресурсу.

Или когда равнозначные потоки имею разные приоритеты.

ПРОБЛЕМЫ КОНКУРЕНТНОГО КОДА - LIVELOCK

СБЕРБАНК ТЕХНОЛОГИИ

Livelock – ситуация, когда поток, который не в blocked состоянии не может прогрессировать дальше, выполняя операцию, которая постоянно не успешна.

СПИСОК ЛИТЕРАТУРЫ

СБЕРБАНК ТЕХНОЛОГИИ

- “Thinking in Java” Bruce Eckel
- “Java Concurrency In Practice”
- “SCJP Sun Certified Programmer for Java 7 Study Guide”

ЗАДАНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Написать свою реализацию ThreadPool.