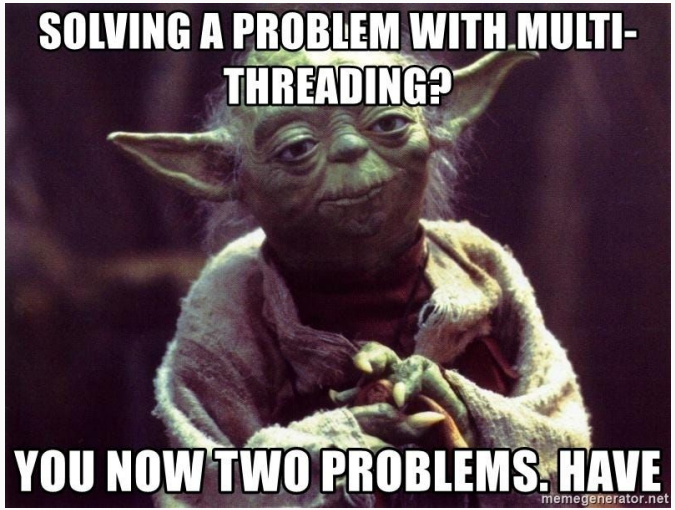
Java BackEnd

Multithreading 2

Andrey Belov | 2025

Проблемы многопоточного кода



ПРОБЛЕМЫ МНОГОПОТОЧНОГО КОДА

1. ГОНКИ
2. ЗАВИСИМОСТЬ ДАННЫХ
3. ВЗАИМНЫЕ БЛОКИРОВКИ
4. ПОТЕРЯ ПРОИЗВОДИТЕЛЬНОСТИ
5. СЛОЖНОСТЬ ОТЛАДКИ

А НУЖЕН ЛИ НАМ МНОГОПОТОЧНЫЙ КОД?

Thread-Safe

ПОТОКОБЕЗОПАСНОСТЬ (Thread-Safe) — это термин, используемый в программировании для обозначения того, что определенный код или объект может безопасно использоваться в многозадачной среде, где выполняются несколько потоков исполнения.

Написание потокобезопасного кода — это, по сути, управление доступом к состоянию и, в частности, к совместному (*shared*) мутируемому состоянию (*mutable state*)

Multithreading: проблемы и решения

Потоки имеют общую память (HEAP), что позволят им эффективно использовать общие данные (shared data), но потенциально может привести к ошибкам:

1. **Взаимо вмешательство потоков (Thread Interference Errors)**
   Несогласованное одновременное изменение общих данных из разных потоков.

   Решение: обеспечить атомарность операции, эксклюзивный доступ к критической секции
   Инструменты: synchronized, Lock, Atomic-классы

2. **Ошибки согласованности памяти (Memory Consistency Errors)**
   Изменение данных внесенные одним потоком, не видны в других потоках. Возможные причины:
   - ➢ Переупорядочивания операций компилятором или процессором (Operations Reordering)
   - ➢ Проблемы когерентности кэша (CPU Cache Coherence problem)

   Решение: установить "happens-before"
   Инструменты: synchronized, volatile, Lock Thread.start()\Thread.join()

КРИТИЧЕСКАЯ СЕКЦИЯ (critical section) – участок кода, который, чтобы избежать ошибок, должен выполняться только одним потоком единовременно, например, код, который обращается к одним и тем же данным через разные потоки.

HAPPENS-BEFORE – гарантия того, что в многопоточном окружении одно действие завершится раньше (“произойдет-до”) другого

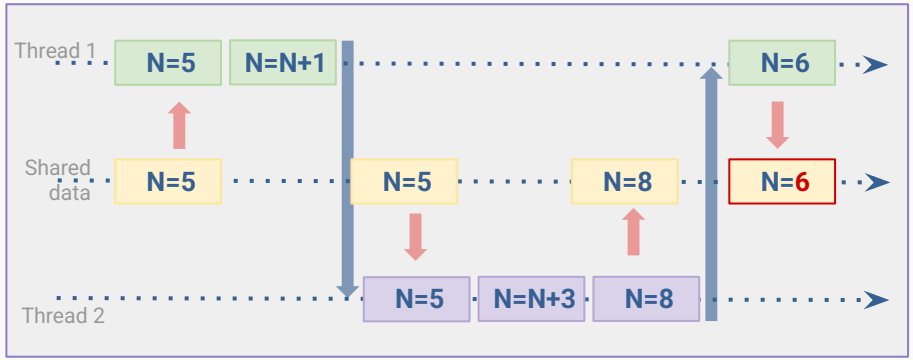
Race Condition

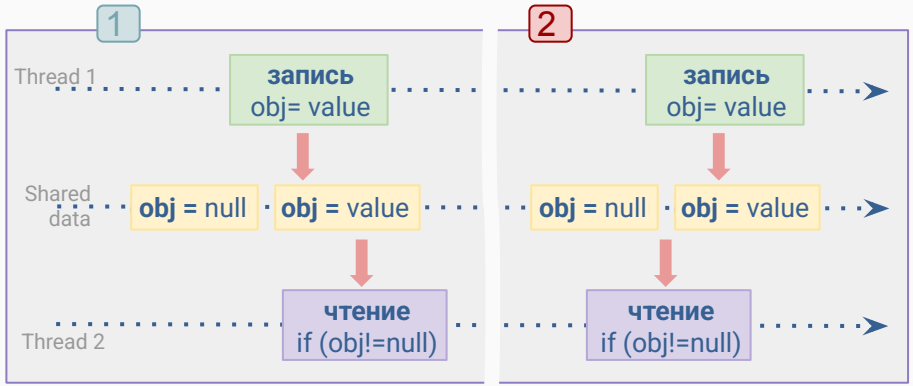
Data Race (*гонка данных*) — это ситуация в многопоточном программировании, когда два или более потока одновременно обращаются к одной и той же переменной, и хотя бы один из потоков изменяет данные, при этом отсутствует механизм синхронизации.

Race Condition (*состояние гонки*) — ситуация, когда несколько потоков конкурируют за один и тот же ресурс, при этом последовательность, в которой осуществляется доступ к ресурсу, влияет на корректность выполнения программы. *Критической секция* — участок кода,который приводит к состоянию гонки.
Результат выполнения программы может быть разным в зависимости от того, кто «прибежит» первым.

1. Read-Modify-Write (чтение–модификация–запись)
2. Check-then-Act (Проверка перед действием)

Race Condition и Data condition





Read-Modify-Write
(прочитать, изменить, записать)

- Один и тот же ресурс (переменная) используется несколькими потоками одновременно
- Итоговое значение является производным от исходного значения
- между чтением и записью может вмешаться другой поток и изменить общее значение.

Причина: не обеспечена атомарность составного действия (compound actions) .

Check-then-Act
(Проверка перед действием)

- Один поток изменяет значение общей переменной
- Другой поток читает значение этой переменной, и в зависимости от прочитанного значения выполняет некоторые действия.
- Т.к. нет гарантии, что обновление значения случилась раньше чем чтение, результат операции не определен.

Причина: чтение зависит от записи, но порядок выполнения не гарантирован.

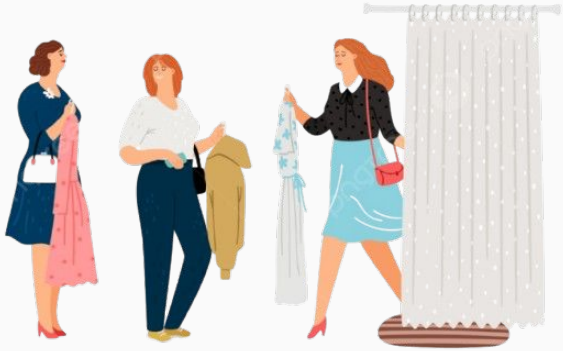
synchronized

synchronized

synchronized не значит “синхронно”,
synchronized значит
“**согласованно**”

synchronized-код выполняется единовременно только одним потоком.

Остальные потоки ожидают (*waiting*) освобождения synchronized-кода



synchronized защищает критическую секцию кода и гарантирует:

- **взаимное исключение** – только один поток может выполнить synchronized секцию в определенный момент времени
- **атомарность** – synchronized секция выполняется целиком, как единая неделимая операция
- **happens-before** – все операции критической секции будут завершены до освобождения synchronized секции.
- **visibility** (видимость) – все потоки смогут увидеть самые последние значения общих изменяемых данных

synchronized

synchronized - метод

МОНИТОР ОБЪЕКТА
this

```
public synchronized void methodName(){

// весь метод – критическая секция

}
```

synchronized блок кода

МОНИТОР ОБЪЕКТА
lock

```
// объект для использования его монитора
private static final Object lock = new Object();

synchronized (lock) {
   // критическая секция
}
```

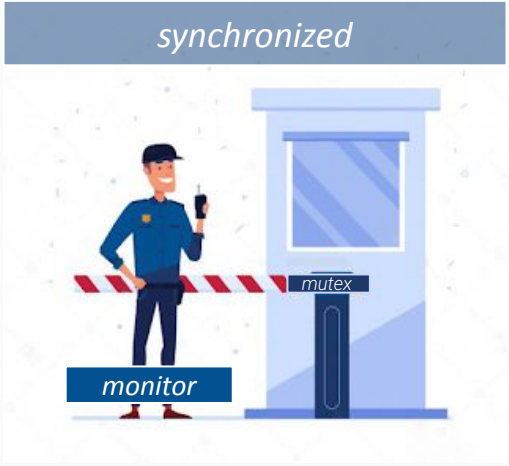
synchronized - статический метод

МОНИТОР ОБЪЕКТА
MyClass.class

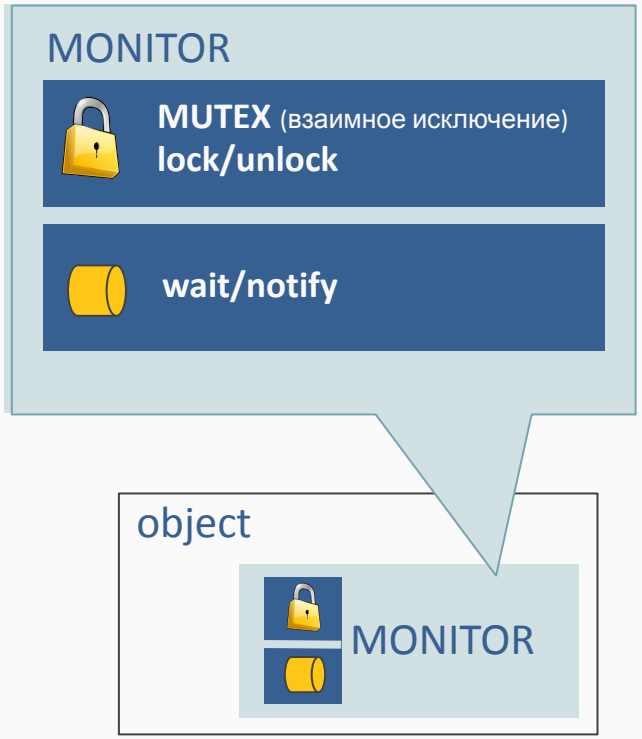
```
public class MyClass {

  public synchronized static voidmethodName(){
      // весь метод – критическая секция
 }
```

synchronized секция всегда связана с объектом-монитором

synchronized использует внутреннюю блокировку (intrinsic lock) объекта, еще называемую блокировкой монитора (monitor lock).



MONITOR



МОНИТОР (monitor) – механизм синхронизации встроенный в объекты Java, который реализует:

- *взаимоисключающий доступ* (Mutex) – только один поток может выполнять *критическую секцию* кода в один момент времени
- механизм ожидания и оповещения потоков (wait/notify).

КАЖДЫЙ ОБЪЕКТ JAVA ИМЕЕТ СВОЙ МОНИТОР.

ЛЮБОЙ ОБЪЕКТ МОЖНО ИСПОЛЬЗОВАТЬ КАК МОНИТОР SYNCHRONIZED БЛОКА

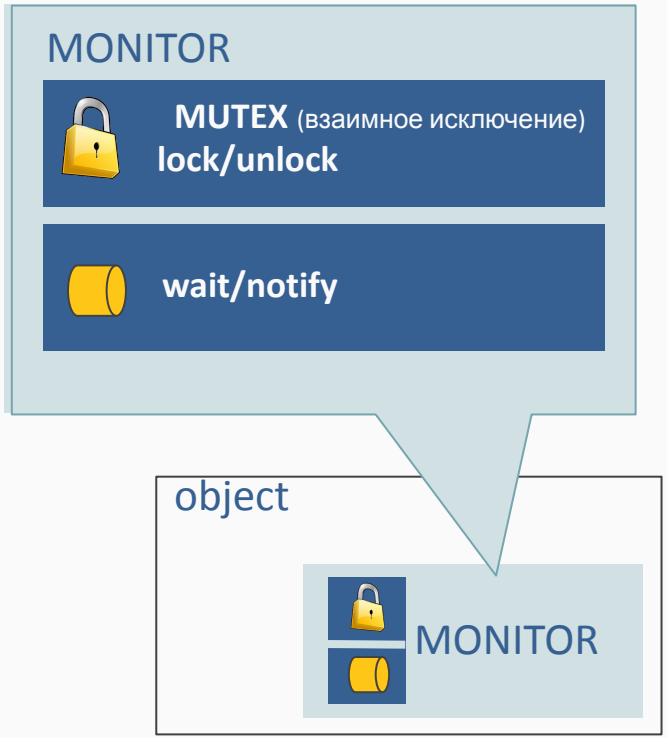
MONITOR

В основе реализации монитора лежит блокировка Mutex

МЬЮТЕКС (Mutex, MUTual EXclusion – взаимное исключение) – системный механизм управления потоками предотвращающий одновременный доступ нескольких потоков к критической секции кода.

- Mutex имеет два состояния: lock (занят), unlock (свободен)
- Mutex никогда не может принадлежать двум разным потокам одновременно.
- Разблокировать Mutex может только тот же поток, который его заблокировал.

MONITOR



- **Каждый объект Java имеет свой монитор**
- В основе реализации монитора лежит MUTEX
- Когда поток входит в synchronized секцию, он “захватывает монитор объекта” (acquire)
- Когда поток выходит из synchronized секции, он “освобождает монитор объекта” (release)
- Только один поток единовременно может захватить монитор, остальные ждут (waiting)
- Поток может уведомить (notify) другие потоки, когда будут выполнены условия, которых они ожидают.
- Поток, владеющий монитором, может захватить монитор повторно, для выполнения другого участка кода, защищенного этим монитором (reentrant)