

Лекция 2

Шаблоны программных платформ языка Java

ФИО преподавателя: Соболев Олег Вадимович

e-mail: sobolev_ov@mail.ru

Тема лекции:

«Многопоточное программирование»

Параллельные процессы

Project Structure: threads, out, src, lec2, Account, Example1, Example2

```

C:\...students\Spring\lec>
1Help 2UserMn 3View 4Edit 5Copy 6RenMov

```

Terminal Output:

Name	Name	Name	Name
ЛК1.pdf	ATIS	home	
ЛК1.pptx	cygwin64	Intel	
ЛК2.pptx	Dell	java	
	GOG Games	PerfLogs	

Диспетчер задач

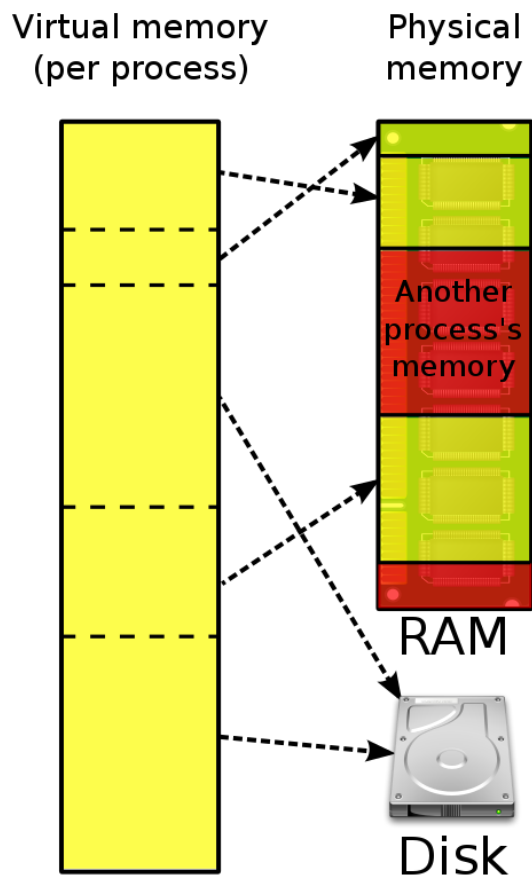
Файл Параметры Вид

Процессы Производительность Журнал приложений Автозагрузка Пользователи Подробности Службы

Имя	ИД п...	Состояние	Имя поль...	ЦП	Память (ак...	Виртуализаци...
igfxEM.exe	7268	Выполняется	User	00	1 676 K	Отключено
IntelCpHDCPSvc.exe	4332	Выполняется	СИСТЕМА	00	196 K	Не разрешено
IntelCpHeciSvc.exe	5480	Выполняется	СИСТЕМА	00	196 K	Не разрешено
java.exe	7960	Выполняется	User	00	4 784 K	Отключено
jhi_service.exe	4116	Выполняется	СИСТЕМА	00	12 K	Не разрешено
LMS.exe	784	Выполняется	СИСТЕМА	00	732 K	Не разрешено
LockApp.exe	2568	Приостановлено	User	00	0 K	Отключено
Lsalso.exe	764	Выполняется	СИСТЕМА	00	0 K	Не разрешено
Isass.exe	864	Выполняется	СИСТЕМА	00	5 704 K	Не разрешено
Microsoft.Photos.exe	9348	Приостановлено	User	00	0 K	Отключено
MoUsCoreWorker.exe	15732	Выполняется	СИСТЕМА	00	404 K	Не разрешено
MsMpEng.exe	4908	Выполняется	СИСТЕМА	00	146 952 K	Не разрешено
NisSrv.exe	7712	Выполняется	LOCAL SER...	00	4 460 K	Не разрешено
OfficeClickToRun.exe	4364	Выполняется	СИСТЕМА	00	3 616 K	Не разрешено
OneApp.JGCC.WinSer...	4252	Выполняется	СИСТЕМА	00	1 624 K	Не разрешено
pacjworker.exe	20476	Выполняется	LOCAL SER...	00	1 560 K	Не разрешено
PanGPA.exe	6724	Выполняется	User	00	3 036 K	Отключено
PanGPS.exe	4396	Выполняется	СИСТЕМА	00	4 116 K	Не разрешено
pg_ctl.exe	4656	Выполняется	NETWORK ...	00	12 K	Не разрешено
pg_ctl.exe	4564	Выполняется	NETWORK ...	00	12 K	Не разрешено
Pinnula.DynamicThe...	1432	Приостановлено	User	00	0 K	Отключено
postgres.exe	6196	Выполняется	NETWORK ...	00	1 040 K	Не разрешено
postgres.exe	6872	Выполняется	NETWORK ...	00	492 K	Не разрешено

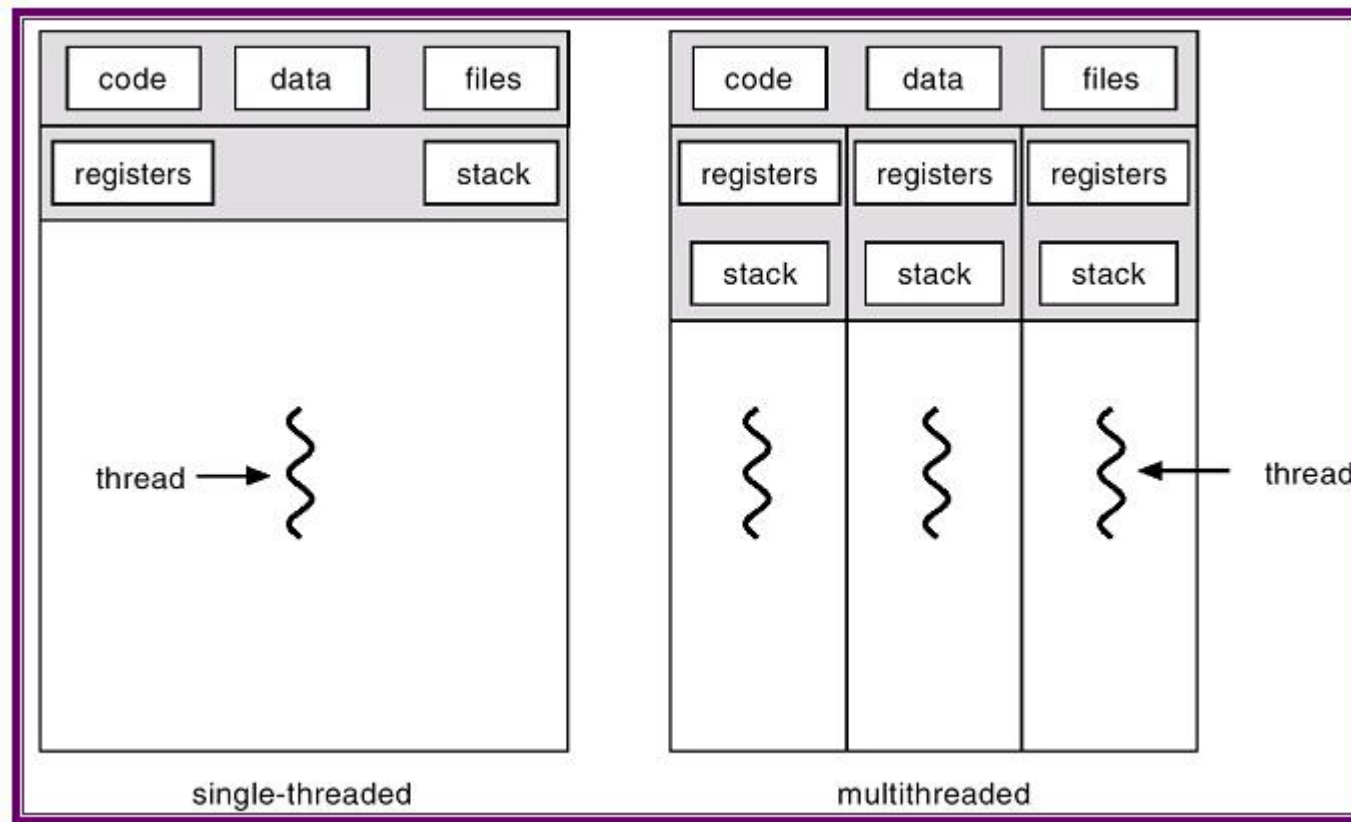
Меньше Снять задачу

Параллельные процессы



Каждый процесс операционной системы имеет свое адресное пространство виртуальной памяти, изолированное от адресных пространств других процессов.

Потоки (threads)



Потоки (threads)

Один процесс может иметь несколько параллельно выполняющихся потоков. В отличие от процессов, все потоки одного процесса выполняются в одном адресном пространстве, т.е. используют общую память.

Потоки (threads)

Потоки используются для:

1. Ускорения работы программы (см. пример с предыдущей лекции: суммирование большого массива данных можно ускорить, если разбить его на два под-массива и суммировать их в отдельных потоках)
2. Систем массового обслуживания (пример: веб-серверы), когда требуется одновременно выполнять много запросов

Thread API

```
public interface Runnable {  
    void run();  
}
```

```
Runnable code = () -> {  
    System.out.println("Hello Runnable!");  
};  
code.run();
```


Thread API

```
Runnable code = () -> {  
    System.out.println("Hello Runnable!");  
};  
Thread t1 = new Thread(code);  
t1.start(); // запуск потока
```

Код, переданный в виде объекта Runnable в конструктор Thread, выполняется в новом потоке параллельно основному потоку программы (main thread).

Thread API

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

```
    Runnable code = () -> {
```

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

```
            System.out.printf("Hello %s from %s\n", i, Thread.currentThread().getName());
```

```
        }
```

```
    };
```

```
    new Thread(code).start(); // выполнение в отдельном потоке
```

```
    code.run(); // выполнение в главном потоке
```

```
}
```

Поток main



Поток t1



Исходный код

```
Hello 0 from main
Hello 1 from main
Hello 2 from main
Hello 3 from main
Hello 0 from Thread-0
Hello 1 from Thread-0
Hello 2 from Thread-0
Hello 3 from Thread-0
Hello 4 from main
Hello 5 from main
Hello 6 from main
Hello 7 from main
Hello 8 from main
Hello 9 from main
Hello 4 from Thread-0
Hello 5 from Thread-0
Hello 6 from Thread-0
Hello 7 from Thread-0
Hello 8 from Thread-0
Hello 9 from Thread-0
```

Thread API

```
public class Thread {  
    void start(); // запуск потока  
    static Thread currentThread(); // поток, который вызвал этот метод  
    String getName(); // имя потока (можно задать через setName)  
    void setName(String name);  
    void join() throws InterruptedException;  
    void setDaemon(boolean on);  
    static void sleep(long millis) throws InterruptedException;  
    State getState();  
}
```

Thread API

Метод `start()` не может быть вызван более одного раза у одного и того же объекта `Thread`. Если вы хотите выполнить один и тот же код в нескольких потоках, создайте для каждого потока свой объект `Thread`.

```
Runnable code = () -> { ... };
```

```
Thread t1 = new Thread(code);
```

```
t1.start();
```

```
t1.start();
```

```
new Thread(code).start(); // запускаем code в еще одном потоке
```

Thread API

Метод `t.join()` ожидает завершение потока `t`:

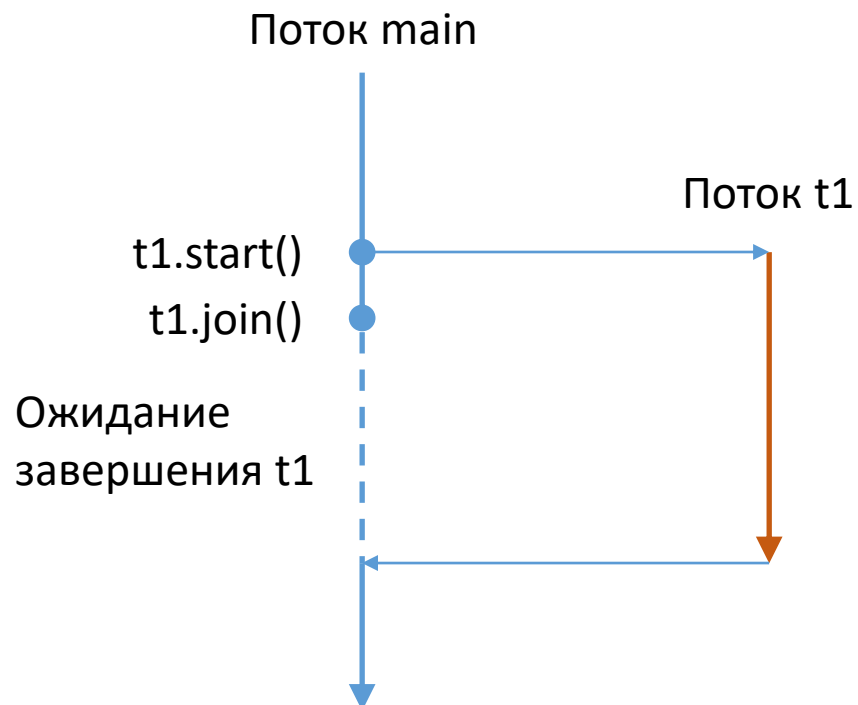
```
Runnable code = () -> { ... };
```

```
Thread t1 = new Thread(code);
```

```
t1.start(); // запуск потока t1
```

```
t1.join(); // главный поток ожидает завершения t1
```

Thread API



Thread API

По умолчанию программа завершается, когда все ее потоки завершаются. Некоторые потоки можно сделать “демонами” (т.е. фоновыми процессами), и тогда они не учитываются при определении необходимости завершения программы.

```
Runnable code = () -> {  
    while (true) { ... }  
};  
Thread t1 = new Thread(code);  
t1.setDaemon(true);  
t1.start(); // запуск потока t1  
// Программа завершается, несмотря на работу потока t1
```

Thread API

Метод `Thread.sleep` останавливает выполнение текущего потока на указанное количество миллисекунд (1/1000 секунды).

```
Thread.sleep(1000); // Ждем 1 секунду
```

Этот метод может генерировать исключение `InterruptedException`. Это исключение используется для остановки потоков; как правило, если вы его поймали, текущий поток нужно завершить.

Thread API

Метод `Thread.getState()`:

- NEW – объект `Thread` был создан, но метод `start` не вызван
- RUNNABLE – поток запущен и работает
- TERMINATED – поток завершен
- BLOCKED – поток ждет освобождения блокировки
- WAITING – поток ждет наступления события без ограничения по времени
- TIMED_WAITING – поток ждет наступления события с ограничением по времени (пример: `Thread.sleep`)

Синхронизация потоков

Потоки прекрасно работают, если они не используют mutable shared state:

- shared – два или более потока обращаются к одним и тем же данным в памяти
- mutable – данные являются изменяемыми

Т.е. все хорошо, если:

- каждый поток работает со своими изменяемыми данными (данные не являются shared)
- потоки работают с общими неизменяемыми данными (данные не являются mutable)

Синхронизация потоков



Синхронизация потоков

```
private static int counter = 0; // mutable shared state (миллионы рублей)
```

```
public static void main(String[] args) throws Exception {
```

```
    Runnable code = () -> {
```

```
        for (int i = 0; i < 5000; i++) {
```

```
            counter++;
```

```
        }
```

```
    };
```

```
    Thread t1 = new Thread(code);
```

```
    Thread t2 = new Thread(code);
```

```
    t1.start();
```

```
    t2.start();
```

```
    t1.join();
```

```
    t2.join();
```

```
    System.out.println(counter);
```

```
}
```

[Исходный код](#)

Результаты:

6356

6059

6233

8872

7362

Синхронизация потоков

Почему так?

“counter++” эквивалентно “counter = counter + 1”:

Поток 1	Поток 2
counter = 0	
counter = counter + 1	counter = counter + 1
counter = 0 + 1	counter = 0 + 1
counter = 1 несмотря на то, что операция ++ выполнена 2 раза	

Из-за того, что “counter++” выполняется параллельно двумя потоками без координации действий потоков, результат может быть случайным. Это то, что называется race condition (гонка).

Синхронизация потоков

```
private static int result = 0;  
private static boolean ready = false;
```

Shared mutable state

[Исходный код](#)

```
public static void main(String[] args) {  
    Runnable producer = () -> {  
        sleep(100);  
        result = 42;  
        ready = true;  
    };  
    Runnable reader = () -> {  
        while (!ready);  
        System.out.println(result);  
    };  
    new Thread(reader).start();  
    new Thread(producer).start();  
}
```

Цикл while никогда не завершается!

Синхронизация потоков

Почему? Компилятор оптимизирует код без специального кода синхронизации в предположении, что код можно менять так, чтобы он оставался корректным с точки зрения одного потока выполнения. Поэтому цикл `while` из примера можно переписать так:

```
boolean localReady = ready;
```

```
while (localReady);
```

1. Доступ к локальной переменной может быть проще оптимизирован
2. Поле `ready` не меняется в теле цикла
3. Компилятор делает вывод, что лучше скопировать значение поля `ready` в локальную переменную

Синхронизация потоков

Так как `localReady` никогда не меняется (несмотря на то, что поле `ready` изменяется), цикл `while` никогда не заканчивается. Т.е. поток `reader` не наблюдает изменений, внесенных потоком `producer` (visibility problem).

Надо отметить, что не все версии Java производят эту оптимизацию и иногда этот код может работать.

Таким образом, компилятору как-то нужно говорить, что к некоторым полям мы обращаемся из разных потоков, чтобы он не производил эту оптимизацию.

Синхронизация потоков

Блок синхронизации:

```
synchronized (lock) { // lock может быть любым объектом  
    // код блока  
}
```

1. Блок синхронизации для одного и того же lock может выполнять одновременно только один поток
2. Если поток 2 выполняет блок синхронизации с lock после того, как поток 1 выполнил блок синхронизации с lock, то поток 2 “увидит” все изменения, внесенные потоком 1.

Синхронизация потоков

Пример:

```
synchronized (lock) {
```

```
    counter++;
```

```
}
```

Поток 1	Поток 2
counter = 0	
counter = counter + 1	(параллельное выполнение невозможно в блоке synchronized)
counter = 0 + 1	
counter = 1	
(параллельное выполнение невозможно в блоке synchronized)	counter = counter + 1
	counter = 1 + 1
counter = 2	

Visibility: Поток 2 гарантированно увидит изменения, внесенные потоком 1

Синхронизация потоков

```
private static int result = 0;  
private static boolean ready = false;
```

[Исходный код](#)

```
public static void main(String[] args) {  
    Object lock = new Object();  
    Runnable producer = () -> {  
        sleep(100);  
        synchronized (lock) {  
            result = 42; ready = true;  
        }  
    };  
    Runnable reader = () -> {  
        while (true) {  
            synchronized (lock) {  
                if (ready) {  
                    System.out.println(result);  
                    break;  
                }  
            }  
        }  
    };  
    new Thread(reader).start(); new Thread(producer).start();  
}
```

Важно, что и у producer, и у reader используется один и тот же объект lock в блоке synchronized. При использовании разных объектов никаких гарантий синхронизации нет.

Гарантируется, что значение поля ready внутри блока synchronized будет прочитано то, которое было записано потоком producer

Синхронизация потоков

Поток producer	Поток reader
result = 0, ready = false	
result = 42	(параллельное выполнение невозможно в блоке synchronized)
ready = true	
result = 42, ready = true	
(параллельное выполнение невозможно в блоке synchronized)	if (ready)
	if (true)
	System.out.println(result);

Visibility: Поток reader гарантированно увидит изменения, внесенные потоком producer

Синхронизация потоков

Важны оба свойства блока `synchronized` – и исключение параллельного выполнения, и обеспечение `visibility`.

В примере `producer/reader` нам более важна `visibility`. Для этого мы можем использовать вместо блоков `synchronized` поля с модификатором `volatile` (см. далее)

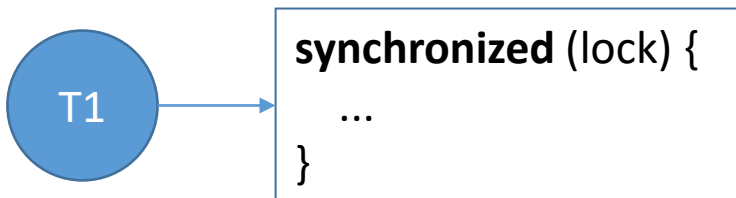
Синхронизация потоков

Блок `synchronized` реализован на основе мониторов:

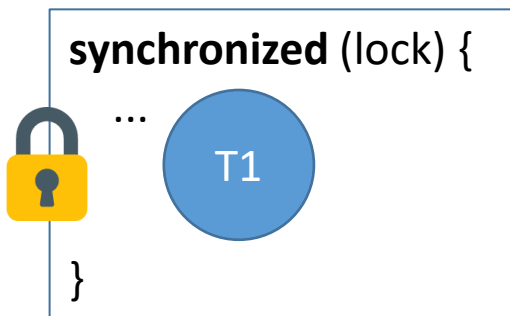
- Монитор может быть в двух состояниях: свободен (`released`) и захвачен (`acquired`)
- При входе в блок `synchronized` происходит попытка захвата монитора:
 - Если монитор свободен, то он становится захвачен
 - Если монитор уже захвачен другим потоком, то текущий поток останавливается и ждет, пока другой поток не освободит монитор
 - Если монитор захвачен текущим потоком, то он остается захвачен
- При выходе из блока `synchronized` монитор освобождается
 - Если при этом другие потоки ждали освобождения этого монитора, то выбирается один из этих потоков, который захватывает монитор и входит в свой блок `synchronized`.

Синхронизация потоков

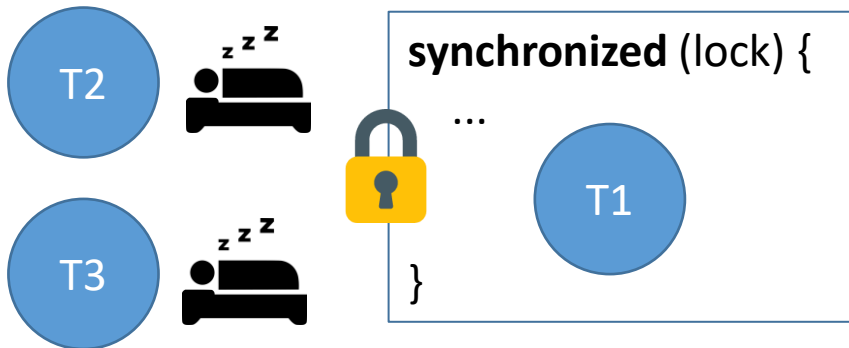
1



2



3

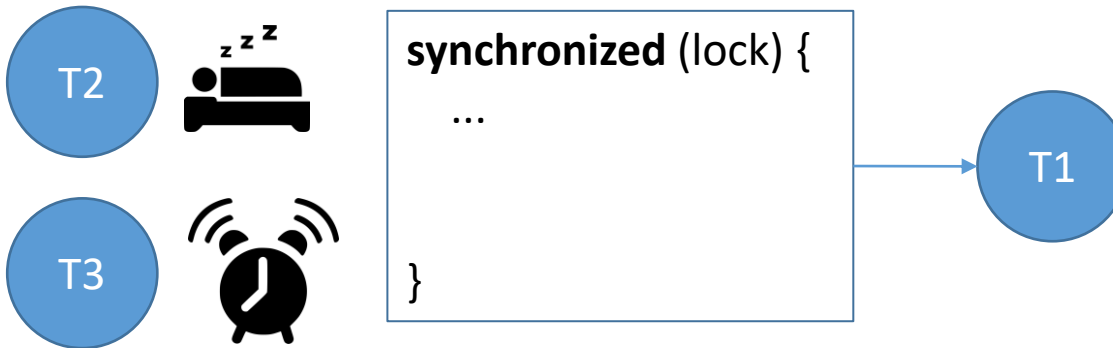


Монитор принадлежит объекту lock, а не конкретному блоку synchronized. Если два или более блока synchronized используют один и тот же lock, то они используют один монитор для блокировки.

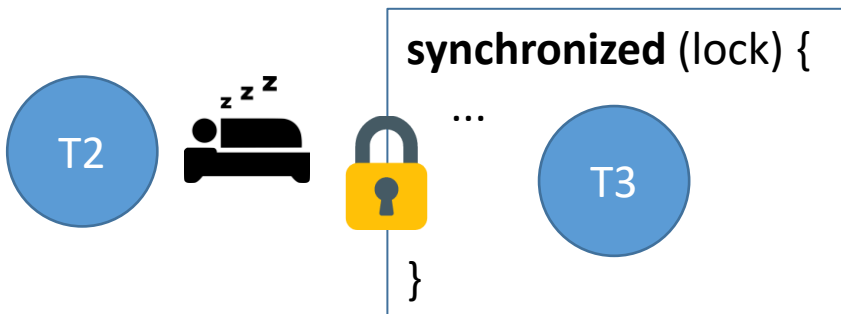
Потоки T2 и T3 находятся в состоянии BLOCKED

Синхронизация потоков

4

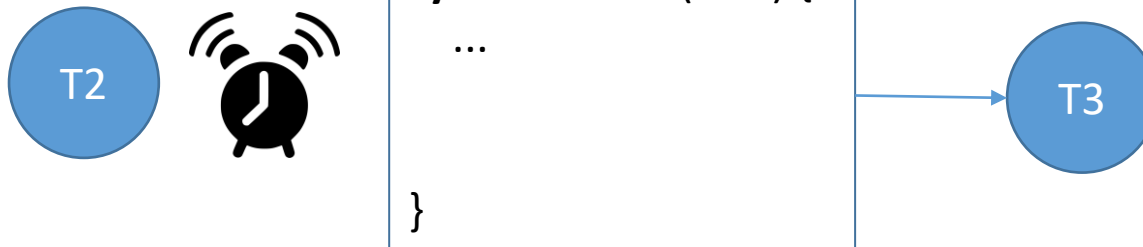


5

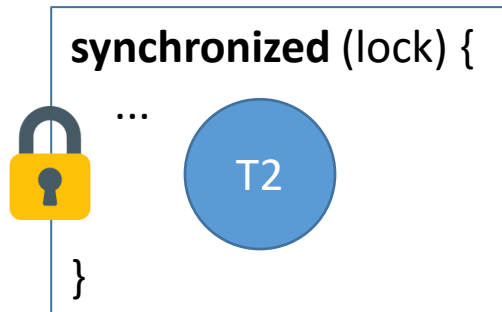


Синхронизация потоков

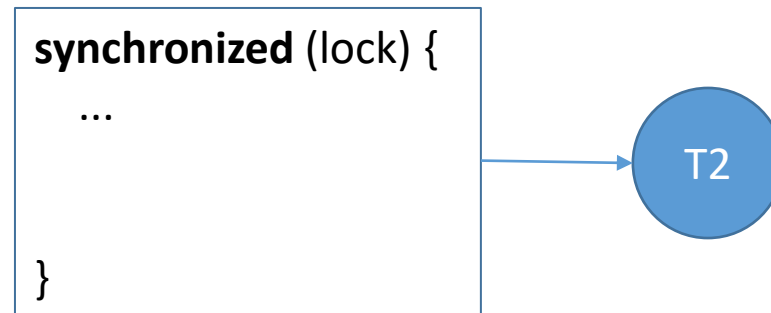
6



7



8



Синхронизация потоков

```
public class Account {  
    private int amount = 100; // Остаток на счете – всегда  $\geq 0$   
    // Снятие sum со счета:  
    public void withdraw(int sum) {  
        if (amount  $\geq$  sum) {  
            amount -= sum;  
        }  
    }  
}
```

Что будет, если 2 потока вызовут withdraw(80)?

Синхронизация потоков

Поток 1	Поток 2
amount = 100	
withdraw(80)	withdraw(80)
if (amount >= 80)	if (amount >= 80)
amount -= 80	
amount = 20	
	amount -= 80
	amount = -60

https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use

Синхронизация потоков

```
public class Account {  
    private int amount = 100; // Остаток на счете – всегда  $\geq 0$   
    private final Object lock = new Object();  
    public void withdraw(int sum) {  
        synchronized (lock) { // гарантирует атомарность выполнения  
            if (amount  $\geq$  sum) {  
                amount -= sum;  
            }  
        }  
    }  
}
```

Синхронизация потоков

Метод является потокобезопасным (thread-safe), если его можно вызывать из нескольких потоков одновременно. Потокобезопасность достигается либо с помощью использования неизменяемых объектов, либо с помощью синхронизации, как в примере Account.

Обычно у класса либо все методы потокобезопасные, либо нет.

Стандартные классы коллекций (ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap) не являются потокобезопасными.

Синхронизация потоков

```
public static void main(String[] args) {  
    Collection<Integer> c = new TreeSet<>();  
    Runnable code = () -> {  
        for (int i = 0; i < 5000; i++) {  
            c.add(i);  
        }  
    };  
    for (int j = 0; j < 100; j++) {  
        Thread t = new Thread(code);  
        t.start();  
    }  
}
```

[Исходный код](#)

Этот код выдает NullPointerException и иногда зацикливается. Это происходит из-за нарушений инвариантов класса, по аналогии с классом Account.

Синхронизация потоков

Правила happens-before (hb):

1. В рамках одного потока любая операция happens-before любой операцией следующей за ней в исходном коде
2. Выход из synchronized блока happens-before входа в synchronized блок на том же мониторе
3. Запись volatile поля happens-before чтение того же самого volatile поля
4. Завершение метода run экземпляра класса Thread happens-before выхода из метода join()
5. Вызов метода start() экземпляра класса Thread happens-before начало метода run() экземпляра того же треда

Синхронизация потоков

Связь happens-before транзитивна, т.е. если X happens-before Y, а Y happens-before Z, то X happens-before Z.

Если X happens-before Y, то все изменения, внесенные до операции X, будут видны в коде, следующем за операцией Y.

Семантика многопоточной работы в Java ([Java Memory Model](https://docs.oracle.com/javase/7/docs/spec/memory-model/)) описывается именно в терминах happens-before.

Синхронизация потоков

Упрощенно можно считать, что при входе в synchronized все переменные считываются из памяти, а при выходе – записываются в память.

На нижнем уровне visibility гарантируется:

- генерацией компилятором кода, записывающего в память содержимого регистров
- инструкцией процессора вида “барьер памяти” (в Intel – MFENCE), которая гарантирует порядок записи данных процессором в память

Синхронизация потоков

Но даже при отсутствии отношения happens-before Java гарантирует, что при чтении поля мы всегда прочитаем значение, записанное одним из потоков, а не мусор.

При этом не гарантируется “свежесть” значения или порядок, в котором мы видим изменения значения. Если сначала поток 1 установил значение поля $x=1$, а потом поток 2 установил значение $x=2$, то поток, который не использует синхронизацию для доступа к полю x , может увидеть любую последовательность значений: только 1; только 2; сначала 1, потом 2; сначала 2, потом 1.

Синхронизация потоков

Модификатор полей volatile:

```
private static volatile boolean ready = false;
```

Правила happens-before гарантируют, что при чтении volatile поля мы читаем последнее записанное значение.

Упрощенно можно представить, что любое обращение к volatile полю завернуто в блок synchronized. Но это не гарантирует атомарности операций! “counter++” все равно подвержено проблеме параллельности даже при наличии volatile. Но для примера producer/reader можно использовать volatile:

Синхронизация потоков

```
private static int result = 0;  
private static volatile boolean ready = false;
```

[Исходный код](#)

```
public static void main(String[] args) {  
    Runnable producer = () -> {  
        sleep(100);  
        result = 42;  
        ready = true;  
    };  
    Runnable reader = () -> {  
        while (!ready);  
        System.out.println(result);  
    };  
    new Thread(reader).start();  
    new Thread(producer).start();  
}
```

- “result=42” hb “ready=true” (п.1)
- “ready=true” hb “while (!ready)” (п.3;
гарантирует, что условие цикла видит
изменение поля)
- “while (!ready)” hb “println(result)” (п.1)

Следовательно,
“result=42” hb “println(result)” (из транзитивности;
гарантирует, что в reader мы увидим присвоенное
в producer значение)

Синхронизация потоков

Пример отсутствия happens-before:

Пример 1

```
// Поток 1:  
int x = 0;  
volatile boolean ready = false;  
  
x = 42;  
ready = true;
```

```
// Поток 2:  
while (true) {  
    // Не гарантируется visibility  
    // для x, так как нет чтения  
    // volatile ready  
    System.out.println(x);  
}
```

Пример 2

```
// Поток 1:  
int x = 0;  
boolean ready = false; // не volatile  
// Компилятор вправе переставить:  
x = 42;  
ready = true;
```

```
// Поток 2:  
while (!ready);  
// Возможные значения:  
// заикливание, 42, 0  
System.out.println(x);
```

Атомарные операции

Для некоторых случаев удобно использовать классы

```
AtomicInteger counter = new AtomicInteger(0);
```

```
// И его аналоги AtomicLong, AtomicBoolean
```

Это аналоги `volatile` полей, но кроме того они добавляют методы, которые выполняются атомарно:

```
int newValue = counter.addAndGet(delta); // Аналог "counter += delta"
```

```
int oldValue = counter.getAndAdd(delta); // Аналог "counter += delta"
```

```
int newValue = counter.incrementAndGet(); // Аналог "++counter"
```

```
int oldValue = counter.getAndIncrement(); // Аналог "counter++"
```

В частности, вместо `"volatile boolean ready"` можно использовать `"AtomicBoolean ready"` (см. [пример](#))

Механизмы синхронизации

	synchronized	AtomicXXX	volatile
Преимущества	Наиболее универсальный механизм синхронизации. Можно задать любой код для атомарного выполнения.	Ограниченный набор атомарных операций	Нет атомарных операций
Недостатки	Наименее эффективный из всех трех. При этом неэффективность проявляется только в случае, когда блок synchronized пытаются выполнить несколько потоков (thread contention).	Промежуточный по эффективности	Наиболее эффективный из трех

Атомарные операции

Вернемся к примеру producer/reader:

```
Object lock = new Object();
Runnable producer = () -> {
    sleep(100);
    synchronized (lock) {
        result = 42;
        ready = true;
    }
};

Runnable reader = () -> {
    while (true) {
        synchronized (lock) {
            if (ready) {
                System.out.println(result);
                break;
            }
        }
    }
};
```

Цикл while будет использовать 100% ядра процессора, не производя полезной работы

Wait/notify

Для подобного кода можно использовать механизм wait/notify:

producer	reader
<pre>synchronized (lock) { ... ready = true; lock.notifyAll(); }</pre>	<pre>synchronized (lock) { lock.wait(); if (ready) { ... } }</pre>

Методы `lock.wait()`, `lock.notify()` и `lock.notifyAll()` можно вызывать только внутри блока “`synchronized (lock)`”.

См. [пример](#) producer/reader с `wait/notifyAll`.

Wait/notify

- Метод `lock.wait()` освобождает монитор `lock` и переводит текущий поток в режим ожидания монитора (состояние `WAITING`). Так как монитор освобожден, другие потоки могут выполнять блоки `"synchronized (lock)"`, пока этот поток находится в режиме ожидания ("спит").
- Метод `lock.notifyAll()` будит все ожидающие этот монитор потоки. Все эти потоки начинают пытаться захватить монитор (это получится не сразу после вызова `notifyAll`, а только когда вызвавший `notifyAll` поток выйдет из блока `synchronized`). После этого все спавшие потоки по очереди захватят монитор и выполнят блок `synchronized`.

Wait/notify

1

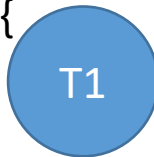


```
synchronized (lock) {  
    ...  
    lock.await();  
    ...  
}
```

2



```
synchronized (lock) {  
    ...  
    lock.await();  
    ...  
}
```



3

```
synchronized (lock) {  
    ...  
    lock.await();  
    ...  
}
```



Wait/notify

4

synchronized (lock) {

...

lock.await();

T1



...

}

T2

synchronized (lock) {

...

lock.notifyAll();

...

}

5

synchronized (lock) {

...

lock.await();

T1



...

}

synchronized (lock) {

...

lock.notifyAll();

...

}

T2

6

synchronized (lock) {

...

lock.await();

T1



...

} Ожидание освобождения монитора

synchronized (lock) {

...

lock.notifyAll();

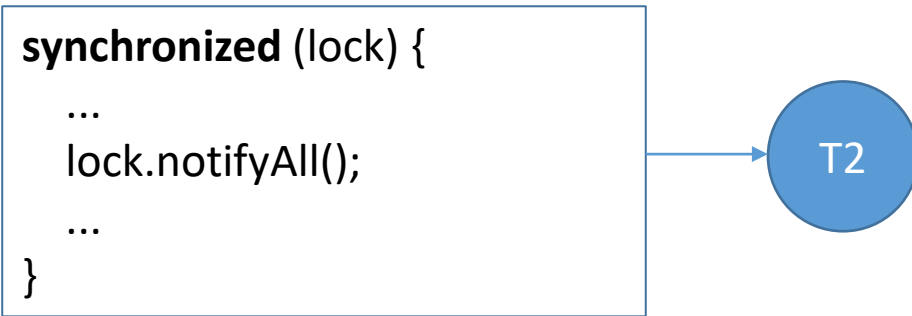
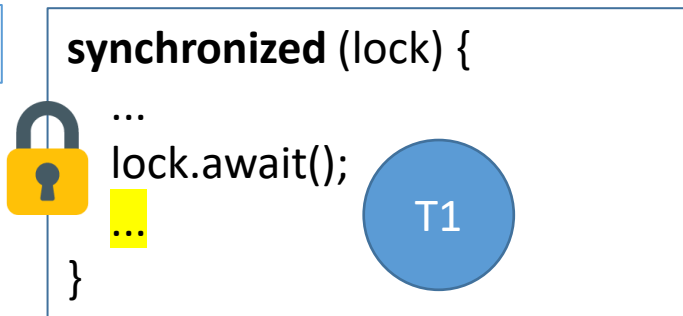
...

}

T2

Wait/notify

7



Метод `lock.notify()`, в отличие от `lock.notifyAll()`, будит только один поток. В нашем случае (и в подавляющем большинстве случаев) нужно использовать `notifyAll`, так как результаты `producer` могут ждать несколько потоков `reader`, и их все нужно уведомить о доступности результата.

Wait/notify

Есть модификация метода `wait` с заданием максимального времени ожидания:

`lock.wait(1000);` // Ждем `notify`, но не более 1 секунды

При этом поток оказывается в состоянии `TIMED_WAITING`.

Wait/notify

Как правило, блок с `wait` проверяет наступление некоторого условия, а блок с `notifyAll` меняет статус этого условия:

```
synchronized (lock) {  
    while (!someCondition()) {  
        lock.wait();  
    }  
    // Условие someCondition выполнено  
}
```

```
synchronized (lock) {  
    setSomeCondition();  
    lock.notifyAll();  
}
```

`lock.wait()` должен вызываться в цикле, так как `wait` может "просыпаться" не обязательно при наступлении условия `someCondition`, так что его нужно перепроверять при каждом пробуждении.

Синхронизация потоков

В пакете `java.util.concurrent.locks` есть классы для расширенной поддержки синхронизации.

Класс `ReentrantLock` является аналогом блока `synchronized` с некоторыми дополнительными функциями. В новых версиях Java рекомендуется использовать его вместо `synchronized`.

Класс `ReentrantReadWriteLock` позволяет блокировать потоки только в случае, когда идет изменение данных, а в случае чтения данных блокировки потоков не происходит.

См. [пример](#) использования `ReentrantLock` и `Condition` вместо `synchronized`.

Синхронизация потоков

До сих пор в примерах мы использовали только один объект для блокировки. В больших программах, как правило, их много. Для чего их нужно много?

```
public class Account {  
    private static final Object GLOBAL_LOCK = new Object();  
    public void withdraw(int sum) {  
        synchronized (GLOBAL_LOCK) { ... }  
    }  
}  
  
Account a1 = new Account();  
Account a2 = new Account();
```

Синхронизация потоков

Если в одном потоке будет вызван `a1.withdraw(80)`, а в другом – `a2.withdraw(80)`, то второй поток будет ждать, пока первый поток выполнит блок `synchronized`, т.к. они используют один объект для блокировки – `GLOBAL_LOCK`.

Но это не нужно (в простейшем случае), так как счета независимы друг от друга; для того, чтобы проверить достаточность средств для снятия со счета `a1`, нам не нужно блокировать счет `a2` – их данные никак не пересекаются.

Поэтому в данном случае можно использовать отдельный объект для блокировки для каждого объекта `Account` (т.е. использовать не статическое поле).

Синхронизация потоков

Как правило, выбирается группа полей, доступ к которым защищается блокировкой. Обычно эта группа полей связана каким-либо инвариантом класса. После этого добавляется поле (либо типа `Object` либо типа `ReentrantLock`) для синхронизации доступа к этой группе полей.

Крайне желательно, чтобы поле для объекта синхронизации было с модификатором `final`.

Синхронизация потоков

К сожалению, потокобезопасность методов не работает в композиции:

```
public void transfer(Account from, Account to, int sum) {  
    if (from.withdraw(sum)) {  
        to.deposit(sum);  
    }  
}
```

Даже если методы `withdraw` и `deposit` потокобезопасны, метод `transfer` уже не является таковым! Здесь проявятся те же проблемы, что и с методом `withdraw` – time to check vs time to use.

Синхронизация потоков

Для корректной работы нам нужно будет заблокировать оба счета перед переводом денег:

```
public void transfer(Account from, Account to, int sum) {  
    synchronized (from.lock) {  
        synchronized (to.lock) {  
            if (from.withdraw(sum))  
                to.deposit(sum);  
        }  
    }  
}
```

Синхронизация потоков

Но если теперь мы выполним в двух потоках:

```
transfer(a1, a2, 10);
```

```
transfer(a2, a1, 10);
```

Поток 1 transfer(a1, a2, 10)	Поток 2 transfer(a2, a1, 10)
synchronized (a1.lock)	synchronized (a2.lock)
монитор a1.lock захвачен потоком 1	монитор a2.lock захвачен потоком 2
synchronized (a2.lock) – ждем освобождения a2	synchronized (a1.lock) – ждем освобождения a1
Взаимная блокировка – deadlock!	

[Исходный код примера](#)

Синхронизация потоков

Возможные решения:

- Использовать одну глобальную блокировку. Неэффективно – потоки блокируются, даже если операция с непересекающимися счетами
- Сортировать счета по какому-либо принципу (если мы берем блокировки все время в одном порядке, deadlock не происходит). Сложно и не всегда возможно.
- Использовать ReentrantLock и взятие блокировки с таймаутом. Нет гарантии, что операция когда-нибудь будет выполнена.

Многопоточное программирование

Итог: многопоточное программирование – это сложно, и по возможности лучше использовать существующие библиотеки и фреймворки для работы с ним.

К счастью, для стандартных приложений потоки напрямую использовать практически не приходится.

Многопоточное программирование

Следующая лекция:

- Прерывание потоков
- Executors и Futures
- Коллекции `java.util.concurrent`
- Дополнительные примитивы синхронизации:
Semaphore, CountDownLatch
- Project Loom