Оглавление

Мотивация

Одновременное выполнение нескольких действий (например, отрисовка пользовательского интерфейса и передача файлов по сети)

Ускорение вычислений (при наличии нескольких вычислительных ядер)

Закон Амдала

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

S — ускорение,

P — доля вычислений, которые возможно распараллелить,

N – количество вычислительных ядер.

Параллелизм в Java

Запуск нескольких JVM на одной или на разных компьютерах (нет общей памяти, взаимодействие через файловую систему или сетевое соединение)

Запуск нескольких потоков внутри JVM (есть общая память, обширная поддержка в языке и стандартной библиотеке)

Проблемы параллельных программ

Гонка (race condition). Ситуация, когда 2 потока пытаются работать с общей памятью (переменными), и один решает что-то изменить в этих данных. Второй поток может этого не увидеть; может увидеть, но не все; может увидеть все, но не в том порядке. Каждая из этих ситуаций может привести к непредсказуемым последствиям.
Для избегания нужно использовать примитивы синхронизации, которые дают эксклюзивный доступ (с каждыми типами данных может работать только один поток).

Взаимная блокировка (deadlock). 1-й поток ждет 2-й ресурс и занимает 1-й ресурс, а 2-й поток занимает 2-й ресурс и ждет 1-й.

java.lang.Thread

Потоки представлены экземплярами класса `java.lang.Thread`. Методы: `String getName()`, `long getId()`, `boolean isDeamon()` (потоки фонового режима, при завершении), `StackTraceElement[] getStackTrace()`, `ThreadGroup getThreadGroup()`.

Tread dump

Список всех потоков с их состояниями и stack trace'ами: в консоли Ctrl+Break

(windows) или Ctrl+\ (linux); jps -l, затем jstack PID; кнопка в IDE (Dump Threads, значок фотоаппарата)

Создание потока

Подкласс Thread

```
1.  public class NewThread extends Thread {
2.
3.    @Override
4.    public void run() {
5.      // ...
6.    }
7.  }
```

```
1.  public class Main {
2.    public static void main(String[] args) {
3.      new NewThread().start();
4.    }
5.  }
```

Runnuble

```
1.  public class Main {
2.
3.    public static void main(String[] args) {
4.      Runnable runnable = () -> ...;
5.      new Thread(runnable).start();
6.    }
7.  }
```

Жизненный цикл потока

Создание объекта `Thread`

Запуск (`thread.start()`)

Работа (выполнение метода `run()`, `thread.isAlive() == true`)

Завершение (завершение метода `run()` или исключение, нельзя перезапусить)

Практика

Пример 1

```
1.  public class Main1 {
2.
3.    public static void main(String[] args) {
4.      for (int i = 0; i < 10; i++) {
5.        Thread thread = new HellowThread();
6.        thread.setName(i + "_thread");
7.        thread.start();
8.      }
9.
10.     System.out.println("Hello from main_thread");
11.   }
12. }
```

```
1.  public class HellowThread extends Thread {
2.
3.    @Override
4.    public void run() {
5.      System.out.println("Hello from " + getName());
6.    }
7.  }
```

Не гарантируется порядок:

```
1.  Hello from 1_thread
2.  Hello from 4_thread
3.  Hello from 2_thread
4.  Hello from 3_thread
5.  Hello from 0_thread
6.  Hello from 5_thread
7.  Hello from main_thread
8.  Hello from 6_thread
9.  Hello from 7_thread
10. Hello from 8_thread
11. Hello from 9_thread
```

```
1.  Hello from 0_thread
2.  Hello from 5_thread
3.  Hello from 2_thread
```

```
Hello from 7_thread
Hello from 1_thread
Hello from 9_thread
Hello from 4_thread
Hello from main_thread
Hello from 3_thread
Hello from 8_thread
Hello from 6_thread
```

Пример 2

```
public class Main2 {

  public static void main() {
    for (int i = 0; i < 10; i++) {
      Thread thread =
          new Thread(new HellowRunnable());
      thread.setName(i + "_thread");
      thread.start();
    }

    System.out.println("Hello from main_thread");
  }
}
```

```
public class HellowRunnable implements Runnable {

  @Override
  public void run() {
    System.out.println(
        "Hello from "
        + Thread.currentThread().getName());
  }
}
```

Пример 3

```
public class Main3 {

  public static void main() {
    for (int i = 0; i < 10; i++) {
      new Thread(
            () -> System.out.println(
              "Hello from"))
          .start();
    }
```

```
10.
11.       System.out.println("Hello from main_thread");
12.     }
13.   }
```

Пример 4

Можно так, но очень осторожно. Нужно понимание, если в этом объекте какое-то состояние, и что будет если 10 потоков будут работать с этим состоянием. Здесь состояния нет, метод `run()` можно совершенно безопасно вызывать из многих потоков одновременно, ничему это не повредит (здесь только чтение из памяти, нет записи в память).

```
 1.  public class Main4 {
 2.
 3.    public static void main() {
 4.      HellowRunnable hellowRunnable =
 5.          new HellowRunnable();
 6.      for (int i = 0; i < 10; i++) {
 7.        Thread thread = new Thread(hellowRunnable);
 8.        thread.setName(i + "_thread");
 9.        thread.start();
10.      }
11.
12.      System.out.println("Hello from main_thread");
13.    }
14.  }
```

```
 1.  public class HellowRunnable implements Runnable {
 2.
 3.    @Override
 4.    public void run() {
 5.      System.out.println(
 6.          "Hello from "
 7.          + Thread.currentThread().getName());
 8.    }
 9.  }
```

Метод run()

Метод `run()` вместо того, чтобы стартовать новый поток, просто исполняет то, что ему передали в текущем потоке, то есть просто исолняет метод, а не создает новый поток.

Прерывание потока

thread.interrupt()

Если поток находится в ожидании (sleep, join, wait), то ожидание прерывается исключением InterruptedException

Иначе у потока просто устанавливается флаг interrupted: флаг проверяется методами interrupted() и isInterrupted(); проверять флаг и завершать поток надо самостоятельно

thread.join() (ожидаем остановки interrupt(), если поток не останавливается, мы можем из него выйти и сделать что-то еще)

Практика

```
1.  public class Main2_1 {
2.
3.    public static void main() throws Exception {
4.      Thread worker = new WorkerThread();
5.      Thread sleeper = new SleeperThread();
6.
7.      System.out.println("Starting threads");
8.      worker.start();
9.      sleeper.start();
```

```
10.
11.     Thread.sleep(100L);
12.
13.     System.out.println("Interrupting threads");
14.     worker.interrupt();
15.     sleeper.interrupt();
16.
17.     System.out.println("Joining threads");
18.     worker.join();
19.     sleeper.join();
20.
21.     System.out.println("All done");
22.   }
23. }
```

```
 1. public class SleeperThread extends Thread {
 2.
 3.   @Override
 4.   public void run() {
 5.     try {
 6.       Thread.sleep(10_000L);
 7.     } catch (InterruptedException e) {
 8.       System.out.println("Sleep unterrupted");
 9.     }
10.   }
11. }
```

```
 1. public class WorkerThread extends Thread {
 2.
 3.   @Override
 4.   public void run() {
 5.     long sum = 0;
 6.     for (int i = 0; i < 1_000_000_000; i++) {
 7.       sum += i;
 8.       if (i % 100 == 0 && isInterrupted()) {
 9.         System.out.println(
10.             "Loop imterrupted at i = " + i +
11.             ", summ = " + sum);
12.         break;
13.       }
14.     }
15.   }
16. }
```

Результат

```
Starting threads
Interrupting threads
Joining threads
Loop imterrupted at i = 27718400, summ =
384154863139200
Sleep unterrupted
All done
```

Если убрать `interrupt()` и `join()`, то программа выполниться и повиснет, т. к. будут выполняться потоки `worker` и `sleeper`

```
public class Main2_2 {

  public static void main() throws Exception {
    Thread worker = new WorkerThread();
    Thread sleeper = new SleeperThread();

    System.out.println("Starting threads");
    worker.start();
    sleeper.start();

    Thread.sleep(100L);

    System.out.println("All done");
  }
}
```

Если сделать потоки `worker` и `sleeper` демонами, то программа завершиться моментально.

```
public class Main2_3 {

  public static void main() throws Exception {
    Thread worker = new WorkerThread();
    worker.setDaemon(true);

    Thread sleeper = new SleeperThread();
    sleeper.setDaemon(true);

```

```
10.      System.out.println("Starting threads");
11.      worker.start();
12.      sleeper.start();
13.
14.      Thread.sleep(100L);
15.
16.      System.out.println("All done");
17.    }
18.  }
```

Возможности встроенной синхронизации

Взаимное исключение (пока один поток что-то делает, другие не могут ему помешать)

Ожидание уведомления (поток ожидает уведомлений от других потоков)

Ключевое слово synchronized

Синхронизованный метод

```
1.  public synchronized void doSomething {
2.
3.    // ...
4.  }
```

Синхронизованный блок внутри метода

```
1.  public void doSomething {
2.    synchronized (obj) {
3.      // ...
4.    }
5.  }
```

(obj) = this или (obj) = Class.class (статический метод)

Ключевое слово synchronized

Синхронизация блоков — по монитору указанного объекта

Синхронизация методов — по монитору текущего объекта (this)

Синхронизация статических методов — по монитору класса

Ожидание и уведомление

Допустимы только внутри `synchronized`

`void wait()`, `void wait(long millis)`, `void wait(long millis, int nanos)`

`void notify()`, `void notyfyAll()`

Практика

Пример 1

```
 1. public class Main3_1 {
 2.
 3.   public static void main() throws Exception {
 4.     Account account = new Account(100_000);
 5.     System.out.println(
 6.         "Begin balance = "
 7.         + account.getBalance());
 8.
 9.     Thread withdrawThread =
10.         new WithdrawThread(account);
11.     Thread depositThread =
12.         new DepositThread(account);
13.
14.     withdrawThread.start();
15.     depositThread.start();
16.
17.     withdrawThread.join();
```

```
    depositThread.join();

    System.out.println(
        "End balance = " + account.getBalance());
  }
}
```

```
public class Account {
  private long balance;

  public Account() {
    this(0L);
  }

  public Account(long balance) {
    this.balance = balance;
  }

  public long getBalance() {
    return balance;
  }

  public void deposit(long amount) {
    checkAmountNonNegative(amount);
    balance += amount;
  }

  public void withdraw(long amount) {
    checkAmountNonNegative(amount);
    if (balance < amount) {
      throw new IllegalArgumentException(
          "not enough money");
    }
    balance -= amount;
  }

  private static void checkAmountNonNegative(
      long amount) {
    if (amount < 0) {
      throw new IllegalArgumentException(
          "negative amount");
    }
  }
}
```

```
public class DepositThread extends Thread {
```

```
  private final Account account;

  public DepositThread(Account account) {
    this.account = account;
  }

  @Override
  public void run() {
    for (int i = 0; i < 20_000; i++) {
      account.deposit(1);
    }
  }
}
```

```
public class WithdrawThread extends Thread {
  private final Account account;

  public WithdrawThread(Account account) {
    this.account = account;
  }

  @Override
  public void run() {
    for (int i = 0; i < 20_000; i++) {
      account.withdraw(1);
    }
  }
}
```

В задаче мы 20_000 раз снимаем и кладем деньги, но по итогу получаем баланс, который отличается от изачального. Мы попадаем на дата рейс (состояние гонки, race condition). Причина в том, что операции инкремент и декремент не атомарны, т. е. один из потоков может положить результат своей работы поверх другого.

```
Begin balance = 100000
End balance = 92534
```

```
Begin balance = 100000
```

```
2.  End balance = 101586
1.  Begin balance = 100000
2.  End balance = 87135
```

Пример 2

Способ решения при помощи ключевого слова `synchronized` над методами изменения счета, т. е. мы говорим, что в один момент времени два потока не могут заходить в эти методы, эти два метода в каждый момент времени выполняться только в одном потоке. Не может такого быть, что мы вошли в метод `deposit()` из одного потока и из другого потока и это один и тот же объект. Кроме того не может быть такого, что мы в метод `deposite()` зашли из одного потока и на том же объекте из другого потока в метод `withdraw()` зашли, потому что мы синхронизируемся по данному экземпляру, но а он общий (если у 2-х потоков одно и та же ссылка).

```
1.  public class Account {
2.    private long balance;
3.
4.    public Account() {
5.      this(0L);
6.    }
7.
8.    public Account(long balance) {
9.      this.balance = balance;
10.   }
11.
12.   public long getBalance() {
13.     return balance;
14.   }
```

```
15.
16.    public synchronized void deposit(long amount) {
17.      checkAmountNonNegative(amount);
18.      balance += amount;
19.    }
20.
21.    public synchronized void withdraw(
22.        long amount) {
23.      checkAmountNonNegative(amount);
24.      if (balance < amount) {
25.        throw new IllegalArgumentException(
26.            "not enough money");
27.      }
28.      balance -= amount;
29.    }
30.
31.    private static void checkAmountNonNegative(
32.        long amount) {
33.      if (amount < 0) {
34.        throw new IllegalArgumentException(
35.            "negative amount");
36.      }
37.    }
38.  }
```

Пример 3

Рекомендуется в `synchronized` запихивать только какой-то маленький кусочек работы, который реально не терпит многопоточного доступа. В данном случае вся полезная работа оказывается в `synchronized` блоке и никакого прироста производительности у нас тут не будет. Но в большинстве реальных ситуаций можно выделить маленький кусочек, который требует синхронизации. Мы маленький кусочек работы программы действительно синхронизируем, обеспечиваем там, чтобы состояние памяти было

консистентно, а другие операции из под этой синхронизации можно вынести,
они исполняются по настоящему параллельно.

```
 1. public class Account {
 2.   private long balance;
 3.
 4.   public Account() {
 5.     this(0L);
 6.   }
 7.
 8.   public Account(long balance) {
 9.     this.balance = balance;
10.   }
11.
12.   public long getBalance() {
13.     return balance;
14.   }
15.
16.   public void deposit(long amount) {
17.     checkAmountNonNegative(amount);
18.     synchronized (this) {
19.       balance += amount;
20.     }
21.   }
22.
23.   public void withdraw(long amount) {
24.     checkAmountNonNegative(amount);
25.     synchronized (this) {
26.       if (balance < amount) {
27.         throw new IllegalArgumentException(
28.             "not enough money");
29.       }
30.       balance -= amount;
31.     }
32.   }
33.
34.   private static void checkAmountNonNegative(
35.       long amount) {
36.     if (amount < 0) {
37.       throw new IllegalArgumentException(
38.           "negative amount");
39.     }
```

```
40.     }
41. }
```

Пример 4

Пример с ожиданием пополнения. Появился еще один метод `waitAndWithdraw()`, который включает в себя ожидание. Это `synchronized` метод, поэтому внутри него можно вызывать `wait()` на текущем объект (на `this`). Здесь мы в цикле проверяем а правда ли, что мы можем нужную операцию выполнить, т. е. достаточно ли на счету денег. Если не достаточно, то отправляем поток в спячку до тех пор, пока его кто-нибудь не пробудет своим `notify()`. Пока этот поток спит данный монитор может быть захвачен каким-нибудь другим потоком. Поэтому метод `deposit()`: вызывает `notifyAll()`. `notifyAll()` рассылает уведомления всем потокам, которые спят на том самом мониторе, на котором мы здесь синхронизировались, т. е. тот же самый экземпляр класса `Account` (`this`). Каждый раз, когда выполняется пополнение счета, после того как поток освобождает метод `deposite()` поток, ждущий в методе `waitAndWithdraw()`, проверяет условие. Если все хорошо, то снимает деньги и завершается, если денег не достаточно, то опять отправляется в спячку.

```
1. public class Main3_4 {
2.
```

```
  3.    public static void main() throws Exception {
  4.      Account account = new Account(0);
  5.
  6.      new DepositThread(account).start();
  7.
  8.      System.out.println(
  9.          "Calling waitAndWithdraw()...");
 10.
 11.      account.waitAndWithdraw(50_000_000);
 12.
 13.      System.out.println(
 14.          "waitAndWithdraw() finished");
 15.    }
 16.  }
```

```
  1.  public class Account {
  2.    private long balance;
  3.
  4.    public Account() {
  5.      this(0L);
  6.    }
  7.
  8.    public Account(long balance) {
  9.      this.balance = balance;
 10.    }
 11.
 12.    public long getBalance() {
 13.      return balance;
 14.    }
 15.
 16.    public synchronized void deposit(long amount) {
 17.      checkAmountNonNegative(amount);
 18.      balance += amount;
 19.      notifyAll();
 20.    }
 21.
 22.    public synchronized void waitAndWithdraw(
 23.            long amount)
 24.          throws InterruptedException {
 25.      checkAmountNonNegative(amount);
 26.      while (balance < amount) {
 27.        wait();
 28.  //      System.out.println("Wakeup: " + balance);
 29.      }
 30.      balance -= amount;
```

```
  }

  private static void checkAmountNonNegative(
        long amount) {
    if (amount < 0) {
      throw new IllegalArgumentException(
          "negative amount");
    }
  }
}
```

```
public class DepositThread extends Thread {
  private final Account account;

  public DepositThread(Account account) {
    this.account = account;
  }

  @Override
  public void run() {
    for (int i = 0; i < 50_000_000; i++) {
      account.deposit(1);
    }
  }
}
```

Модель памяти

Модель памяти — спецификация того, как будут работать ваши программы, в случае, если запущенно несколько потоков. Более конкретно: какие операции являются атомарными, какие нет (17 глава Java language спецификейшн). Второе важное свойство — видимость, когда состояние сделанные одним потоком видны в другом.

Атомарные операции

Чтение и запись полей всех типов, кроме long и double, происходит атомарно

Если поле объявлено с модификатором `volatile`, то атомарно читаются и пишутся даже `long` и `double`

Видимость

Изменения значений полей, сделанные одним потоком, могут быть не видны в другом потоке

Изменения, сделанные одним потоком, могут быть видны в другом потоке в ином порядке

Правила формализованы при помощи отношения happens-before (отношение: если в одном потоке произошло некое событие x, а в другом потоке произошло событие z, то мы гарантировано знаем, что все произошло до x, будет видно после z; вопрос в том, что это за пары таких событий, и как их нами обеспечить)

Семантика final

happens-before

Запись `volatile`-поля happens-before чтения этого поля

Освобождение монитора happens-before захват того же монитора

`thread.start()` happens-before `thread.run()`

Завершение `thread.run` happens-before выход из `thread.join()`

Практика

Пример 1

В многопоточной ситуации эта программа не корректна, потому что метод `getInstance()` не синхронизованный (одновременной зайдут несколько потоков и создадут объект).

```
 1. public class Singleton1 {
 2.   private int foo;
 3.   private String bar;
 4.
 5.   private Singleton1() {
 6.     this.foo = 13;
 7.     this.bar = "zap";
 8.   }
 9.
10.   private static Singleton1 instance;
11.
12.   public static Singleton1 getInstance() {
13.     if (instance == null) {
14.       instance = new Singleton1();
15.     }
16.     return instance;
17.   }
18. }
```

Пример 2

Способ решения проблемы при помощи ключевого слова `synchronized`. Некоторым здесь не нравиться, что приходиться платить накладные расходы за синхронизацию на каждый вызов `getInstance()`, в то время

как реально хотелось бы заплатить только первый раз, когда реально происходит инициализация.

```
 1. public class Singleton2 {
 2.   private int foo;
 3.   private String bar;
 4.
 5.   private Singleton2() {
 6.     this.foo = 13;
 7.     this.bar = "zap";
 8.   }
 9.
10.   private static Singleton2 instance;
11.
12.   public synchronized static
13.         Singleton2 getInstance() {
14.     if (instance == null) {
15.       instance = new Singleton2();
16.     }
17.     return instance;
18.   }
19. }
```

Поэтому некоторые светлые умы начинают изобретать такую конструкцию:

```
 1. public class Singleton3 {
 2.   private int foo;
 3.   private String bar;
 4.
 5.   private Singleton3() {
 6.     this.foo = 13;
 7.     this.bar = "zap";
 8.   }
 9.
10.   private static Singleton3 instance;
11.
12.   public synchronized static
13.       Singleton3 getInstance() {
14.     if (instance == null) {
15.       synchronized (Singleton3.class) {
16.         if (instance == null) {
```

```
17.              instance = new Singleton3();
18.            }
19.          }
20.        }
21.        return instance;
22.      }
23.  }
```

Т. е. если instance создан, то мы не пойдем в instance блок, не будем платить за синхронизацию. Это идиома даже получила специальное название - антипаттерн Double checked locking (двойная проверка и лок). Она не корректна. На вопрос почему может помочь знание о модели памяти Java. В частности нужно понимать, что инициализация (instance = new Singleton3()) это не только присвоение ссылки на новый объект (new Singleton3()) в поле instance, но это еще присвоение значений полей, создаваемого экземпляра. Эти записи присвоения полей экземпляра (в конструкторе) и присвоение этой статической переменной (17) они могут быть произвольным образом переупорядочены. Мы уже записали в это поле instance ссылку на новый объект, при этом его значения полей еще не инициализированы. И получится, что ссылка на недостроенный объект утекла в нашу программу. Кто-то на нем может вызвать какие-то методы и начать получать странные исключения из-за того, что состояние объекта некорректно.

Поэтому этот случай ошибочный, за исключение такого случая:

```
 1. public class Singleton4 {
 2.   private int foo;
 3.   private String bar;
 4.
 5.   private Singleton4() {
 6.     this.foo = 13;
 7.     this.bar = "zap";
 8.   }
 9.
10.   private static volatile Singleton4 instance;
11.
12.   public synchronized static
13.       Singleton4 getInstance() {
14.     if (instance == null) {
15.       synchronized (Singleton4.class) {
16.         if (instance == null) {
17.           instance = new Singleton4();
18.         }
19.       }
20.     }
21.     return instance;
22.   }
23. }
```

`volatile` нам обеспечивает happens-before между записью в `volatile` ссылки на объект (17). Мы знаем, что присвоение полей должны были произойти до нее, до записи этой ссылки в объект. А в (14) мы делаем чтение из `volatile` переменной, соответственно гарантировано видим все то, что в памяти происходило в (17).

В лучшем случаем используйте второй способ (`volatile` метод).

Источники

https://www.youtube.com/watch?v=zxZ0BXlTys0&ab_channel=ComputerScienceCenter