|  |  |
| --- | --- |
| ДИСЦИПЛИНА | **Разработка клиент-серверных приложений** |
|  | (полное наименование дисциплины без сокращений) |
| ИНСТИТУТ | **информационных технологий** |
| КАФЕДРА | **Инструментального и прикладного программного обеспечения** |
|  | (полное наименование кафедры) |
| ВИД УЧЕБНОГО | **Материалы для практических/семинарских занятий** |
| МАТЕРИАЛА | (в соответствии с пп.1-11) |
| ПРЕПОДАВАТЕЛЬ | **Куликов Александр Анатольевич**  **Зарипов Евгений Андреевич** |
|  | (фамилия, имя, отчество) |
| СЕМЕСТР | **7, 2023-2024** |
|  | (указать семестр обучения, учебный год) |



# Практическая работа №1

**по дисциплине «Разработка клиент-серверных приложений» для студентов 4 курса. Форма обучения: Очная.**

Тема: Примитивы синхронизации и их использование при создании клиент- серверных приложений

# Теоретическое введение Многопоточность в Java

Встроенная поддержка многопоточного программирования языком про- граммирования Java является его удобной отличительной особенностью. Созда- ние многопоточных приложений не только ключ к решению многих задач,но и практика ускорения и улучшения алгоритмов существующих решений. Единица такого приложения называется потоком(Thread).

При этом очень важно разделять многозадачность, основанную напроцес- сах и многозадачность, основанную на потоках. Для иллюстрации разницы обра- тимся к определению процесса. Процесс - выполнение пассивных инструкций компьютерной программы на процессоре ЭВМ. Но более простыми словами, процесс по сути является программой, которая выполняется на компьютере. Ей выделены определенные ресурсы и мощности ЭВМ. То есть процесс — это сово- купность кода и данных, разделяющих общее виртуальное адресное простран- ство. Простой пример многозадачности на процессах — это несколько одновре- менно запущенных программ на одном и том же компьютере,например, веб-брау- зер, текстовый редактор и интегрированная среда разработки.Подробнее о про- цессах можно узнать из данной статьи.

Когда речь идет о многозадачности, основанной на потоках, то это выпол- нение нескольких задач внутри одного тяжеловесного процесса. При реализации приложений, поддерживающих многозадачность, обычно используется многоза- дачность, основанная на процессах, по ряду причин:

1. Потоки используют одно адресное пространство.
2. Коммуникации между потоками являются экономичными, а переключение контекста между потоками характеризуется низкой стоимостью
3. Многозадачностью на основе потоков возможно управлять средствами языка программирования java

Но существуют некоторые сложности и ограничения c использованиемпо- токов:

1. На некоторых платформах запуск новых потоков может замедлить работу приложения
2. Для каждого потока создается свой собственный стек памяти. Значит, чем больше потоков, тем больше памяти используется одним процес- сом.
3. Во многих системах есть не только ограничение на количество памяти, но и на количество потоков

# Класс Thread

Управление потоками в java начинается с класса Thread, представляющего сущность потока. Официальная документация по данному классу находится по ссылке. Стоит сразу понимать, что экземпляры класса Thread в Java сами по себе не являются потоками. Это лишь своего рода API для низкоуровневых потоков, которыми управляет JVM и операционная система. Когда при помощи java launcher'а мы запускаем JVM, она создает главный поток с именем main и ещё несколько служебных потоков. Существует 2 типа потоков: демоны и не демоны. Демон-потоки — это фоновые потоки (служебные), выполняющие какую-тора- боту в фоне. Важно это понимать для того, что VM продолжает выполнение про- граммы (процесса), до тех пор, пока не вызван метод Runtime.exit() иливсе не де- мон-потоки завершили свою работу успешно или с ошибкой. Что значит,что про- грамма завершит свою работу при работе фоновых потоков. Подробнее про де- мон-потоки можно прочитать здесь.

Чтобы создать новый поток, нужно создать экземпляр класса Thread. Дан- ный поток будет являться дочерним по отношению к главному потоку main. Для определения текущего потока класс Thread имеет статический метод Thread.currentThread(), использование данного метода проиллюстрировано в Листинге 1.

Листинг 1 – Использование метода получения имени текущего потока public static void main(String[] args)

{

Thread t = Thread.currentThread(); // получаем главныйпоток Sys- tem.out.println(t.getName()); // main

}

Следуя документации, есть 2 способа создания потока: создать класс- наследник от Thread или класс, реализующий интерфейс Runnable. Первый спо- соб представлен в Листинге 2.

Листинг 2 – Реализация класса-потомка Thread public class Main {

public static void main(String[] args) {

MyThread myThread = new MyThread("myThread");myThread.start();

}

}

class MyThread extends Thread{ MyThread(String name){

super(name);

}

public void run(){

System.out.printf("%s started... \n", Thread.currentThread().getName()); try{

Thread.sleep(500);

}

catch(InterruptedException e){

System.out.println("Thread has been

interrupted");

}

System.out.printf("%s finished... \n", Thread.currentThread().getName());

}

}

Описывается класс MyThread, наследуемый от класса Thread. В это классе

определяется конструктор, принимающий название потока, которое передается в конструктор родительского класса. Также переопределен метод run(), который отвечает за выполнение логики потока. Для того, чтобы запустить поток исполь- зуется метод start().

# Методы синхронизации и управления потоками

Подробнее про синхронизацию потоков можно узнать в статье.

Поток в процессе своего выполнения может засыпать. Это самой простой тип взаимодействия с другими потоками.

В операционной системе, на которой установлена виртуальная Java ма- шина, где выполняется Java код, есть свой планировщик потоков, называемый Thread Scheduler. Именно он решает, какой поток, когда запускать. Иллюстрация усыпления потока представлена в Листинге 2.

Прерывание потоков также является важной операцией при их использова- нии. Данный метод Thread.interrupt() используется дляинформирования потока о том, что ему нужно прерваться. Для обработки ситуации прерывания потока вы- брасывается исключение InterruptedException, обработка которого позволяет «от- ловить» момент прерывания. Пример представлен в Листинге 2.

При запуске потока в примере выше Main thread завершался до дочернего потока. Как правило, более распространенной ситуацией является случай, когда Main thread завершается самым последним. Для этого надо применить метод join(). В этом случае текущий поток будет ожидать завершения потока, для кото- рого вызван метод join. Пример использования метода представлен в Листинге 3.

Листинг 3 – Пример использования метода join

public static void main(String[] args) { System.out.println("Main thread started..."); MyThread t= new MyThread("MyThread "); t.start();

try{

t.join();

}

catch(InterruptedException e){ System.out.printf("%s has been inter- rupted",

t.getName());

}

System.out.println("Main thread finished...");

}

Метод join() заставляет вызвавший поток (в данном случае Main thread) ожидать завершения вызываемого потока, для которого и применяется метод join (в данном случае MyThread). Если в программе используется несколько дочерних потоков, и надо, чтобы Main thread завершался после дочерних, то для каждого дочернего потока надо вызвать метод join. Данный метод является простейшим методом синхронизации потоков. Для этих целей в java существует также меха- низм, именуемый монитор. С каждым объектом ассоциирован некоторый мони- тор, а потоки могут его заблокировать "lock" или разблокировать "unlock".

Подробнее можно узнать из документации, а также здесь и здесь. Раз- беремпростой пример, представленный в Листинге 4.

Листинг 4 – Пример применения монитора public class HelloWorld{

public static void main(String []args){Object object = new

Object(); synchronized(object) { System.out.println("Hello World");

}

}

}

Действие, обернутое в ключевое слово synchronized является «Захватом мо- нитора» или «Получением лока» для объекта object в потоке main. нет соперни- чества (т.е. никто больше не хочет выполнить synchronized по такому жеобъекту), Java может попытаться выполнить оптимизацию, называемую "biased locking". В заголовке объекта в Mark Word(метаданные невидимые и недоступные програм- мисту) выставится соответствующий тэг и запись о том, к какому потоку привя- зан монитор. Это позволяет сократить накладные расходы при захватывании мо- нитора.

Если монитор уже ранее был привязан к другому потоку, тогда такой бло- кировки недостаточно. JVM переключается на следующий тип блокировки — basic locking. Она использует compare-and-swap (CAS) операции. При этом в за- головке в Mark Word уже хранится не сам Mark Word, а ссылка на его хранение

+ изменяется тэг, чтобы JVM поняла, что у нас используется базовая блокировка. Если же возникает соперничество (contention) за монитор нескольких пото- ков (один захватил монитор, а второй ждёт освобождение монитора), тогдатэг в Mark Word меняется, и в Mark Word начинает храниться ссылка уже намо- нитор как объект — некоторую внутреннюю сущность JVM. Как сказано в JEP,в таком случае требуется место в Native Heap области памяти на хранение этой сущности. Ссылка на место хранения этой внутренней сущности и будет

находиться в Mark Word объекта.

Таким образом, как мы видим, монитор — это действительно механизм обеспечения синхронизации доступа нескольких потоков к общим ресурсам. Су- ществует несколько реализаций этого механизма, между которымипереключа- ется JVM.

Рассмотрим пример, когда происходит ожидание по монитору. Пример представлен в Листинге 5.

Листинг 5 – Пример с блокированием объекта и ожиданием по монитору public static void main(String[] args) throws

InterruptedException {

Object lock = new Object();

Runnable task = () -> { synchronized (lock) {

System.out.println("thread");

}

};

Thread th1 = new Thread(task);th1.start(); synchronized (lock) {

for (int i = 0; i < 8; i++) { Thread.currentThread().sleep(1000);Sys- tem.out.print(" " + i);

}

System.out.println(" ...");

}

}

В примере главный поток сначала отправляет задачу task в новый поток, а потом сразу же "захватывает" лок или монитор и выполняет с ним долгую опера- цию (8 секунд). Всё это время task не может для своего выполнения зайти вблок synchronized, т.к. лок(монитор) уже занят. Следует также обратить вниманиена ре- ализацию потока Task, выполненную в функциональном стиле, за счет реализа- ции интерфейса Runnable. Данная возможность доступна с Java 1.8. Еслипоток не может получить лок, он будет ждать этого у монитора. Как только получит — продолжит выполнение.

Кроме блоков синхронизации может быть синхронизирован целый метод. В одну единицу времени данный метод будет выполняться только одним пото- ком. В случае методов объекта локом(то, что блокируется) будет выступатьthis. Если метод статический, то локом будет не this (т.к. для статического методане может быть this), а объект класса (Например, Integer.class).

Существуют случаи, когда монитор какого-либо объекта или ресурса за- хвачен и требуется явно дождаться его высвобождения для работы с ним. Для этого существует метод Thread.wait. Выполняется метод wait на объекте, на мо- ниторе которого требуется выполнить ожидание. Рассмотрим пример, представ- ленный в Листинге 6.

Листинг 6 – Пример использования метода wait public class Program {

public static void main(String[] args) { Store store=new Store();

Producer producer = new Producer(store);Consumer con- sumer = new Consumer(store); new Thread(pro- ducer).start();

new Thread(consumer).start();

}

}

// Класс Магазин, хранящий произведенные товарыclass Store{ private int product=0;

public synchronized void get() {while (prod- uct<1) {

try {

wait();

}

catch (InterruptedException e) {

}

}

product--;

System.out.println("Покупатель купил 1 товар"); Sys- tem.out.println("Товаров на складе: " +

product);

notify();

}

public synchronized void put() {while (prod- uct>=3) {

try {

wait();

Продолжение листинга 6

}

catch (InterruptedException e) {

}

}

product++;

System.out.println("Производитель добавил 1

товар");

System.out.println("Товаров на складе: " + product);

notify();

}

}

// класс Производитель

class Producer implements Runnable{Store store; Producer(Store store){

this.store=store;

}

public void run(){

for (int i = 1; i < 6; i++) {store.put();

}

}

}

// Класс Потребитель

class Consumer implements Runnable{Store store; Consumer(Store store){

this.store=store;

}

public void run(){

for (int i = 1; i < 6; i++) {store.get();

}

}

}

В примере определен класс магазина, потребителя и покупателя. Произво- дитель в методе run() добавляет в объект Store с помощью его метода put() 6 то- варов. Потребитель в методе run() в цикле обращается к методу get объекта Store для получения этих товаров. Оба метода Store - put и get являются синхронизиро- ванными.

Для отслеживания наличия товаров в классе Store проверяется значение пе- ременной product. По умолчанию товара нет, поэтому переменная равна 0. Метод get() - получение товара должен срабатывать только при наличии хотя быодного товара. Поэтому в методе get проверяется, отсутствует ли товар.

Если товар отсутствует, вызывается метод wait(). Этот метод освобождает монитор объекта Store и блокирует выполнение метода get, пока для этого же мо- нитора не будет вызван метод notify(), который продолжает работу ротока, у ко- торого ранее был вызван метод wait().

Когда в методе put() добавляется товар и вызывается notify(), то метод get() получает монитор и выходит из конструкции while (product<1), так как товар до- бавлен. Затем имитируется получение покупателем товара. Для этого выводится сообщение, и уменьшается значение product: product--. И в конце вызов метода notify() дает сигнал методу put() продолжить работу.

В методе put() работает похожая логика, только теперь метод put() должен срабатывать, если в магазине не более трех товаров. Поэтому в цикле проверяется наличие товара, и если товар уже есть, то освобождается монитор с помощью wait() и ожидается вызова notify() в методе get().

Таким образом, с помощью wait() в методе get() ожидается, когда произво- дитель добавит новый продукт. А после добавления вызывается notify(),инфор- мируя, что на складе освободилось одно место, и можно еще добавлять.

А в методе put() с помощью wait() ожидается освобождения места наскладе. После того, как место освободится, добавляется товар и через notify() уведомля- ется покупатель о том, что он может забирать товар.

# Методы параллельного программирования в Java

Существует еще одна возможность реализации потока с помощьюинтер- фейса java.util.concurrent.Callable доступного с версии 1.5. Его отличием отинтер- фейса Runnable, описанного ранее, является то, что в отличие от Runnable, новый интерфейс объявляет метод call, который возвращает результат. Кроме того, по умолчанию он выбрасывает исключение. Простой пример реализации данного интерфейса представлен в Листинге 7.

Листинг 7 – Task

Callable task = () -> { return "Hello, World!";

};

Для чего нужна задача, которая возвращает результат? Это использу- ется,

когда нужно получить результат действий, которые будут в дальнейшем ис- пользоваться.

Интерфейс java.util.concurrent.Future описывает API для работы с зада- чами,результат которых мы планируется получить в будущем: методы получе- ния результата, методы проверки статуса.

Одной из реализаций интерфейса Future является реализация java.util.concurrent.FutureTask. То есть это Task, который будет выполнен во Future. Рассмотрим пример в Листинге 8.

Листинг 8 – Пример использования FutureTask public class HelloWorld {

publicstatic void main(String []args) throws Exception

{

Callable task = () -> { return "Hello, World!";

};

FutureTask<String> future = newFutureTask<>(task);

new Thread(future).start(); System.out.println(future.get());

}

}

В примере проиллюстрировано получение при помощи метода get

результата из задачи task. Важно, что в момент получения результата при по- мощиметода get выполнение становится синхронным.

Типовое создание и старт нового потока можно вынести в отдельныйме- тод, реализуя при этом специальный интерфейс Executor. Пример представленв Листинге 9.

Листинг 9 – Пример использования интерфейса Executor public static void main(String []args) throws Exception {

Runnable task = () -> System.out.println("Taskexecuted"); Executor executor = (runnable) -> {new

Thread(runnable).start();

};

executor.execute(task);

}

Подробнее можно узнать в данной статье.

# WorkStealingPool и ForkJoin Framework

Существует проблема простаивания одних потоков, когда другие пере- гружены задачами. Существует решение Work Stealing — это такой алгоритмра- боты, при котором простаивающие потоки начинают забирать задачи других потоков или задачи из общей очереди. Пример необходимости рассмотрим в Листинге 10.

Листинг 10 – Пример необходимости Work Stealing public static void main(String[] args) {Object lock = new

Object();

ExecutorService executorServ-

ice =Executors.newCachedThreadPool();

Callable<String> task = () -> {

System.out.println(Thread.currentThread().getName());lock.wait(2000); System.out.println("Finished");return "result";

};

for (int i = 0; i < 5; i++) { executorService.submit(task);

}

executorService.shutdown();

}

В примере ExecutorService создаст 5 потоков, которые встанут в очередь по

локу объекта lock. Что поменяется если заменить Executors.newCachedThreadPool на Executors.newWorkStealingPool()? Задачи выполнятся не в 5 потоков, а меньше. В случае со StealingPool потоки не будут

вечно простаивать в wait, они

начнут выполнять соседние задачи. Такая реализация доступна за счет исполь- зования демон-потоков.

Внутри WorkStealingPool используется технология ForkJoinPool или fork/join framework. Fork/JoinPool оперирует в своей работе таким понятием как java.util.concurrent.RecursiveTask. Также есть аналог

— java.util.concurrent.RecursiveAction. RecursiveAction не возвращают результат. Таким образом RecursiveTask похож на Callable, а RecursiveAction похож на Runnable. Метод fork запускает асин- хронно в отдельном потоке некоторую задачу. А метод join позволяет до- ждаться завершения выполнения работы. Подробнее про технологию можно узнать здесь. Видеодоклад здесь.

# Задание на практическую работу №1

**Задание 1**

Дан массив из 10000 элементов. Необходимо написать несколько реализа- ций некоторой функции F в зависимости от варианта. Функция должна быть ре- ализована следующими способами:

1. Последовательно
2. С использованием многопоточности (Thread, Future, и т. д.)
3. С использованием ForkJoin.

После каждой операции с элементом массива (сравнение, сложение)до- бавить задержку в 1 мс при помощи Thread.sleep(1);

Провести сравнительный анализ затрат по времени и памяти при за- пускекаждого из вариантов реализации.

Варианты функций (выбор варианта осуществляется по формуле «Номер всписке группы % 3»)

1. Поиск суммы элементов массива.
2. Поиск максимального элемента в массиве.
3. Поиск минимального элемента в массиве.

# Задание 2

Программа запрашивает у пользователя на вход число. Программа ими- тирует обработку запроса пользователя в виде задержки от 1 до 5 секунд выво- дит результат: число, возведенное в квадрат. В момент выполнения запроса пользователь имеет возможность отправить новый запрос. Реализовать с ис- пользованием Future.

# Задание 3

Реализовать следующую многопоточную си- стему.Файл. Имеет следующие характеристики:

1. Тип файла (например XML, JSON, XLS)
2. Размер файла — целочисленное значение от 10 до 100.

Генератор файлов -- генерирует файлы с задержкой от 100 до 1000 мс.

Очередь — получает файлы из генератора. Вместимость очереди — 5 файлов.

Обработчик файлов — получает файл из очереди. Каждый обработчик имеет параметр — тип файла, который он может обработать. Время обработки файла: «Размер файла\*7мс»

Система должна удовлетворять следующими условиям:

1. Должна быть обеспечена потокобезопасность.
2. Работа генератора не должна зависеть от работы обработчиков, и наоборот.
3. Если нет задач, то потоки не должны быть активны.
4. Если нет задач, то потоки не должны блокировать другие потоки.
5. Должна быть сохранена целостность данных.