Київський національний університет імені Тараса Шевченка

Факультет комп`ютерних наук та кібернетики

Кафедра інтелектуальних програмних систем

Операційні Системи

Індивідуальне завдання

Варіант: Блокуючі запити, Future, Unix-сокет, Java

Виконав студент 3-го курсу

Групи ІПС-32

Кот Андрій Анатолійович

Київ-2024

Завдання

Завдання полягало в вивченні методів роботи Java Networking, тобто сокети, протоколи, серверну та клієнтську частину. Розробити програми, яка в свою чергу реалізовувала алгоритми блокування запитів, Future та Unix-сокети.

Теорія

1. **Блокуючі запити**

Блокуючий запит — це операція, яка зупиняє виконання потоку до завершення цієї операції. Потік «блокується» і чекає, поки операція завершиться або отримає результат.

Характеристики:

- **Синхронність**: Блокуючі запити виконуються синхронно, тобто наступний код не виконується, поки запит не завершиться.
- **Простота реалізації**: Блокуючі операції прості у використанні, оскільки не вимагають складних механізмів для обробки результатів.
- **Негативний вплив на продуктивність**: Якщо потік заблокований, він не може виконувати інші завдання, що може знижувати продуктивність багатопотокових додатків.

2. **Future**

Future — це об'єкт в Java, який представляє результат асинхронної операції, яка може завершитися в майбутньому.

Основні характеристики:

- Future дозволяє отримати результат обчислення, яке виконується в іншому потоці.

- Асинхронний підхід: основний потік може продовжувати працювати, поки результат не стане доступним.
- Підтримує методи для перевірки стану завдання, очікування завершення та скасування.

Основні методи Future:

- **get()**: Блокує виконання, поки результат не буде готовий.
- **isDone()**: Повертає true, якщо завдання завершене.
- **cancel(boolean mayInterruptIfRunning)**: Скасовує виконання завдання.
- **isCancelled()**: Повертає true, якщо завдання було скасоване.

3. **Unix-сокети**

Unix-сокети — це механізм міжпроцесної взаємодії (Inter-Process Communication, IPC) для обміну даними між процесами на одній машині. Вони використовують файлову систему для встановлення адреси сокета і працюють безпосередньо через ядро операційної системи.

Типи Unix-сокетів:

1. **SOCK_STREAM**: Використовує потоки даних (аналог TCP).
2. **SOCK_DGRAM**: Використовує датаграми (аналог UDP).
3. **SOCK_SEQPACKET**: Послідовні пакети, схожі на потоки, але зберігається структурованість пакетів.

Як працюють Unix-сокети:

- Сервер створює сокет і прив’язує його до адреси у файловій системі (наприклад, /tmp/socket).
- Клієнт підключається до цього сокета через ту ж адресу.
- Процеси обмінюються даними через сокет, використовуючи механізми читання і запису.

Реалізація

Почнемо з **блокуючих запитів**.

Як вже було з'ясовано, блокуючим запитом вважається довільна сихронізована команда, яка блокує потік до тих пір поки, запит не буде повністю виконаний. Тому цим можна вважати навіть звичайний readline() з BufferedReader. Але якщо ми говоримо про потоки, тоді можемо привести даний приклад:

```
public static void main(String[] args){
   //Створюємо довільний об'єкт, який буде блокуватися
   Resource resource = new Resource();
   //Розміщуємо об'єкт в двух потоках
   MyThread myThread1 = new MyThread(resource);
   MyThread myThread2 = new MyThread(resource);
   //Запускаємо
   myThread1.start();
   myThread2.start();
}
```

В середині кожного потоку перевизначаємо метод run()

```
@Override
public void run() {
   try{
       System.out.println("Thread: "+this.getName()+", Start");
       // Блокуємо об'єкт
       synchronized(resource){
           System.out.println("Thread: "+this.getName()+", block resource");
           sleep(5000);
       }
       System.out.println("Thread: "+this.getName()+", Unblock resource");
   }
```

```
    catch (InterruptedException e){
        System.out.println(e);
    }
}
```

Коли ми зупускаємо програму один з потоків блокує ресурс, тим самим інший поток, який звертається до цього самого об’єкту, переходить у стан очікування, поки перший його не звільнить.

Дані з консолі:

```
Thread: Thread-1, Start
Thread: Thread-0, Start
Thread: Thread-1, block resource
Thread: Thread-1, Unblock resource
Thread: Thread-0, block resource
Thread: Thread-0, Unblock resource
```

Теперер перейдемо до Future.

Future, це клас який дозволя викликати методи в іншому потоці, не перешкоджаючи роботі основного потоку, а також отримувати результат пізніше у вигляді якихось даних (якщо викликати метод .get() тоді поток виклику блокується). Все це можливо, тому що ми використовуємо інтерфейс Callable, а не Runable, який дозволяє повертати змінну, як результат.

Перейдемо до прикладу:

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

    //Створюємо ExecutorService, щоб отримати пул потоків

    ExecutorService executorService = Executors.newCachedThreadPool();

    int x = 1, y = 2;

    //Імплементуємо локальний таск за допомогою лямбда виразу (Callable це функціанальний інтерфейс)

    Callable<Integer> task = () -> {

        Thread.sleep(2000);

        return x + y;

    };

    // Розташовуємо такс у future

    Future<Integer> future = executorService.submit(task);

    try {

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

        // Викликаємо метод

        int res = future.get();

        System.out.println("Result is: " + res);


        System.out.println("Done");

    } catch (Exception e) {

        System.out.println(e);

    } finally {

        executorService.shutdown();

    }

}
```

Дані з консолі:

```
Call future...
Result is: 3
Done
```

Щодо до **Unix-сокетів**, спочатку потрібно пояснити, що це взагалі таке. На відміну від звичайних сокетів, які потребують IP адреси та порт, для підключення, будь то в локальній чи серверній мережі, юнікс-сокети працюють лише в локальній та за допомогою файлової системи, що набагато швидше порівнянно з попереднім.

Приклад серверної частини:

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

```
   //Створюємо файл у папці, до якої буде відключений сервер

   UnixSocketAddress address = new UnixSocketAddress(new
File("/tmp/unix_socket_java"));

   try (UnixServerSocketChannel serverChannel =
UnixServerSocketChannel.open()) {

       // Біндимо файл для нашого сокета

       serverChannel.configureBlocking(true);

       serverChannel.socket().bind(address);

       System.out.println("Server is waiting for a client...");

       //  Відкриваємо канал та очікуємо на клієнта

       try (UnixSocketChannel clientChannel = serverChannel.accept()) {

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

           ByteBuffer buffer = ByteBuffer.allocate(1024);

           // Тепер поки наш клієнт передає дані, та залишається на зв'язку,
ми продовжуємо виводити повідомлення отримані від нього

           while (true) {

               buffer.clear();
```

```
                int bytesRead = clientChannel.read(buffer);


                if (bytesRead == -1) {
                    //Клієнт відключився
                     System.out.println("Client disconnected");
                    break;
                }


                buffer.flip();
                String message = new String(buffer.array(), 0,
buffer.limit());
                System.out.println("Received from client: " + message);


                if ("End".equals(message.trim())) {
                    System.out.println("Ending connection as 'End' received");
                    break;
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Приклад клієнтської сторони:

```
public static void main(String[] args) {
    SocketAddress socketAddress =
UnixDomainSocketAddress.of("/tmp/unix_socket_java");
    System.out.println("Connecting to server... ");
    // Спроба підключитися до сервера
    try(SocketChannel socketChannel = SocketChannel.open(socketAddress)){
```

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

        BufferedReader userInput = new BufferedReader(new
InputStreamReader(System.in));

        String line="";

        // Тепер можемо передавати повідомлення

        while(!line.equals("End")){

            line= userInput.readLine();

            ByteBuffer buffer = ByteBuffer.wrap(line.getBytes());

            socketChannel.write(buffer);

        }


    }

    catch(IOException e){

        System.out.println(e);

    }

}
```

Дані з консолі:

Запускаємо програму Сервер:

```
Server is waiting for a client...
```

Запускаємо Клієнтську частину:

```
Connecting to server...
Connected
```

```
Server is waiting for a client...
Client connected
```

Тепер можемо передавати повідомлення через клієнта:

Client:

```
Connecting to server...
Connected
Hello Server
Goodbye
End

Process finished with exit code 0
```

Server:

```
Server is waiting for a client...
Client connected
Received from client: Hello Server
Received from client: Goodbye
Received from client: End
Ending connection as 'End' received

Process finished with exit code 0
```

Висновок:

У ході роботи було вивчено та реалізовано механізми блокуючих запитів, асинхронної взаємодії за допомогою `Future`, а також Unix-сокетів для міжпроцесної комунікації. Було створено сервер і клієнт, які взаємодіють через Unix-сокет, використовуючи бібліотеку

JNR-UNIXSOCKET. Це дозволило дослідити переваги швидкості та безпеки локальних сокетів, а також порівняти їх із традиційними мережевими механізмами. Отримані результати демонструють, що кожен із розглянутих підходів має свої сильні сторони, а вибір оптимального залежить від специфіки задачі, особливо в контексті локальних і розподілених систем.