NAMA : Fachri Yuda Arvianto

NIM : 362241011

Teknik Informatika

Tugas Modul 3 Pemograman Jaringan

1. Model blocking dalam pemrograman socket mengacu pada cara program berinteraksi dengan soket (socket) saat melakukan operasi I/O (Input/Output). Dalam konteks ini, "blocking" berarti bahwa operasi I/O akan memblokir eksekusi program sampai operasi tersebut selesai.

   Konsep utama dari model blocking adalah bahwa saat sebuah program melakukan operasi I/O pada soket, program akan berhenti atau "memblokir" di titik tersebut sampai data yang diinginkan tersedia atau operasi I/O selesai. Ini berarti bahwa program tidak akan melanjutkan eksekusi kode selanjutnya sampai operasi I/O selesai, sehingga dapat menyebabkan program terhenti atau tidak responsif jika waktu yang dibutuhkan untuk operasi I/O cukup lama.

   Contoh penggunaan model blocking dalam pemrograman socket dapat dijelaskan dengan contoh sederhana menggunakan Java dan kelas Socket dari paket java.net. Berikut adalah contoh sederhana klien-server menggunakan model blocking:

   Contoh Server (Blocking Server)

```
import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class BlockingServer {
    public static void main(String[] args) {
        try {
            // Membuat server socket yang mendengarkan koneksi di port 8888
            ServerSocket serverSocket = new ServerSocket(8888);

            System.out.println("Server listening on port 8888...");

            // Menunggu koneksi dari klien
            Socket clientSocket = serverSocket.accept();

            // Mendapatkan input stream dari koneksi
            InputStream inputStream = clientSocket.getInputStream();

            // Membaca data dari input stream (blok hingga data tersedia)
            byte[] buffer = new byte[1024];
            int bytesRead = inputStream.read(buffer);

            // Menampilkan data yang diterima dari klien
```

```
            System.out.println("Received: " + new String(buffer, 0, bytesRead));

            // Menutup koneksi
            clientSocket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Contoh Client (Blocking Client)

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;

public class BlockingClient {
    public static void main(String[] args) {
        try {
            // Membuat koneksi ke server yang berjalan di localhost dan port 8888
            Socket socket = new Socket("localhost", 8888);

            // Mendapatkan output stream dari koneksi
            OutputStream outputStream = socket.getOutputStream();

            // Mengirim data ke server (blok hingga data terkirim)
            String message = "Hello, Server!";
            outputStream.write(message.getBytes());

            // Menutup koneksi
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. Perbedaan Model Blocking dan Non-Blocking

   Model Blocking:

   Keuntungan:

   - Sederhana: Lebih mudah untuk diimplementasikan dan dipahami karena eksekusi program berhenti atau "memblokir" sampai operasi I/O selesai.
   - Sinkronus: Eksekusi program berjalan secara sinkronus, memudahkan pemrograman dan pemahaman alur program.

Kerugian:

- Kinerja: Dapat menyebabkan program menjadi tidak responsif jika operasi I/O memakan waktu lama, karena program akan terhenti sampai operasi selesai.\
- Skalabilitas: Sulit untuk menangani banyak koneksi secara efisien karena setiap koneksi dapat memblokir eksekusi program.

Mode Non-Blocking:

Keuntungan:

- Responsif: Program tetap responsif terhadap input dan dapat melanjutkan eksekusi bahkan ketika operasi I/O tertentu sedang berlangsung.
- Skalabilitas: Dapat menangani banyak koneksi secara lebih efisien karena tidak ada koneksi yang memblokir eksekusi program.

Kerugian:

- Kompleksitas: Lebih sulit untuk diimplementasikan dan memerlukan manajemen sumber daya lebih lanjut, seperti polling atau callback.
- Sinkronisasi Manual: Memerlukan pengelolaan sinkronisasi manual untuk menangani pemrosesan data yang mungkin belum sepenuhnya tersedia.

3. Operasi yang biasanya menggunakan model blocking socket dan tiga operasi yang menggunakan model non-blocking socket.

   Model Blocking Socket:

   1) Socket.accept(): Operasi ini digunakan pada sisi server untuk menerima koneksi dari klien. Program akan memblokir hingga klien berhasil terhubung.
   2) InputStream.read(byte[]): Ketika membaca dari input stream (contohnya, dari socket input stream), operasi ini akan memblokir hingga data tersedia atau koneksi ditutup.
   3) OutputStream.write(byte[]): Ketika menulis ke output stream (contohnya, ke socket output stream), operasi ini akan memblokir hingga data berhasil dituliskan atau koneksi ditutup.

   Model Non-Blocking Socket:

   1) SocketChannel.accept(): Operasi ini digunakan pada sisi server untuk menerima koneksi dari klien dalam model non-blocking. Jika tidak ada koneksi yang tersedia, operasi ini akan mengembalikan nilai null tanpa memblokir.
   2) SocketChannel.read(ByteBuffer): Menggunakan SocketChannel untuk membaca dari socket dengan model non-blocking. Operasi ini akan mengembalikan jumlah byte yang sebenarnya dibaca, tetapi tidak akan memblokir eksekusi program.
   3) SocketChannel.write(ByteBuffer): Menggunakan SocketChannel untuk menulis ke socket dengan model non-blocking. Operasi ini akan mengembalikan jumlah byte yang sebenarnya ditulis, tetapi tidak akan memblokir eksekusi program.

4. Contoh aplikasi yang mungkin lebih baik menggunakan satu model daripada yang lain.

   Model Blocking:

   1) Aplikasi yang Sederhana dan Mudah Dipahami:
      - Contoh: Aplikasi sederhana dengan satu atau sedikit koneksi, seperti aplikasi konsol yang menyediakan layanan berbasis teks.
   2) Kesesuaian dengan Model Sinkronus:
      - Contoh: Aplikasi yang tidak memerlukan banyak koneksi bersamaan dan di mana sinkronisasi operasi I/O lebih mudah dipelajari dan diimplementasikan.
   3) Ketersediaan Sumber Daya Terbatas:
      - Contoh: Dalam lingkungan dengan sumber daya terbatas (misalnya, perangkat keras dengan keterbatasan daya komputasi), model blocking dapat lebih mudah dikelola.

   Model Non-Blocking:

   1) Kinerja dan Responsivitas Tinggi:
      - Contoh: Aplikasi yang membutuhkan kinerja tinggi dan responsivitas yang baik, seperti server jaringan yang menangani banyak koneksi bersamaan.
   2) Skalabilitas untuk Banyak Koneksi:
      - Contoh: Server aplikasi dengan banyak klien bersamaan, seperti server web yang harus menangani banyak permintaan HTTP secara simultan.
   3) Pengolahan Acara dan Kejadian Bersamaan:
      - Contoh: Aplikasi yang merespons kejadian atau pesan tanpa harus menunggu selesainya operasi I/O, seperti aplikasi grafis atau permainan yang memproses input dari pengguna.

5. Untuk menghindari "busy waiting" atau polling yang berputar terus menerus dalam implementasi model non-blocking, pendekatan yang umum digunakan adalah menggunakan ‘Selector’ dalam bahasa pemrograman yang mendukung non-blocking I/O, seperti Java.

   Dalam Java, ‘Selector’ digunakan bersama dengan SelectableChannel (seperti SocketChannel atau ServerSocketChannel) untuk memonitor beberapa kanal dan menentukan mana yang siap untuk operasi tertentu (seperti membaca atau menulis). Dengan menggunakan ‘Selector’, program dapat menunggu hingga satu atau lebih kanal siap untuk operasi I/O tanpa perlu melakukan polling secara aktif.

   Berikut adalah langkah-langkah umum untuk menggunakan ‘Selector’ dalam Java:

   1. Membuat Selector:

      Selector selector = Selector.open();

   2. Mendaftarkan Kanal ke Selector:

Menggunakan metode register() untuk mendaftarkan kanal dengan selector dan menentukan jenis operasi yang diminati (seperti SelectionKey.OP_READ atau SelectionKey.OP_WRITE).

```
SocketChannel socketChannel = //... buat SocketChannel
socketChannel.configureBlocking(false);
SelectionKey key = socketChannel.register(selector, SelectionKey.OP_READ);
```

3. Memantau kunci yang siap:
   Menggunakan metode select() untuk memantau kanal-kandal yang sudah didaftarkan dengan selector.

   ```
   int readyChannels = selector.select();
   ```

4. Mengambil kunci yang siap untuk diolah:
   Menggunakan selectedKeys() untuk mendapatkan kumpulan kunci yang siap untuk operasi I/O.

   ```
   Set<SelectionKey> selectedKeys = selector.selectedKeys();
   ```

5. Mengolah kunci yang siap:
   Iterasi melalui kumpulan kunci dan menangani operasi I/O yang sesuai.

   ```
   for (SelectionKey key : selectedKeys) {
     if (key.isReadable()) {
       // Menangani operasi membaca
     } else if (key.isWritable()) {
       // Menangani operasi menulis
     }
   }
   ```