Modul 8

Thread Programming

Tugas Pendahuluan :

1. Apa yang dimaksud dengan "Thread" dalam konteks pemrograman? Jelaskan perbedaan antara single-threaded dan multi-threaded.
2. Mengapa penggunaan thread diperlukan dalam pengembangan perangkat lunak? Berikan setidaknya dua contoh situasi di mana penggunaan thread dapat meningkatkan kinerja atau responsivitas aplikasi.
3. Jelaskan perbedaan antara "process" dan "thread". Bagaimana sebuah thread berhubungan dengan proses dalam suatu aplikasi?

Materi :

Thread programming adalah teknik pemrograman di mana program dapat menjalankan beberapa alur atau proses secara bersamaan, yang disebut dengan thread. Thread adalah unit kecil dari pemrosesan yang berjalan di dalam suatu proses, dan masing-masing thread dapat memiliki aliran eksekusi yang terpisah satu sama lain.

Dalam thread programming, sebuah program dapat dipecah menjadi beberapa thread, masing-masing thread melakukan tugas yang berbeda. Dengan cara ini, program dapat melakukan beberapa tugas secara bersamaan, yang dapat mengoptimalkan penggunaan sumber daya komputer.

Terdapat beberapa konsep yang penting dalam thread programming, yaitu:

1. Thread creation: proses pembuatan thread baru, yang dilakukan dengan memanggil fungsi tertentu pada level sistem operasi.

2. Thread scheduling: proses pengaturan dan pengalokasian sumber daya CPU untuk setiap thread yang berjalan, yang dapat dilakukan secara pre-emptive atau cooperative.

3. Thread synchronization: proses sinkronisasi antara thread yang berjalan, untuk menghindari kondisi race dan deadlock. Hal ini dapat dilakukan dengan menggunakan beberapa teknik, seperti mutex, semaphore, dan monitor.

4. Thread communication: proses komunikasi antara thread yang berjalan, yang dapat dilakukan dengan menggunakan shared memory, message passing, atau socket.

Thread programming dapat digunakan dalam berbagai aplikasi, seperti aplikasi multimedia, aplikasi web, aplikasi database, dan aplikasi game. Namun, perlu diingat bahwa thread programming juga memiliki beberapa tantangan, seperti debugging yang sulit dan overhead yang tinggi. Oleh karena itu, penggunaan thread programming harus dipertimbangkan dengan hati-hati, dan harus dipahami dengan baik oleh para pengembang.

Thread programming dapat sangat berguna dalam pemrograman jaringan, karena memungkinkan program untuk melakukan beberapa operasi jaringan secara bersamaan. Beberapa contoh penggunaan thread programming dalam pemrograman jaringan meliputi:

1. Multi-threaded server: sebuah server dapat diimplementasikan menggunakan thread programming, dengan setiap koneksi klien ditangani oleh thread yang berbeda. Dengan cara ini, server dapat melayani beberapa koneksi klien secara bersamaan, yang dapat meningkatkan kinerja server.
2. Multi-threaded client: sebuah client juga dapat diimplementasikan menggunakan thread programming, dengan setiap koneksi jaringan ditangani oleh thread yang berbeda. Dengan cara ini, client dapat melakukan beberapa operasi jaringan secara bersamaan, seperti mengirim permintaan dan menerima respons dari server.
3. Background thread: sebuah program jaringan juga dapat menggunakan thread programming untuk menjalankan tugas-tugas jaringan di latar belakang, seperti memperbarui informasi dari server secara berkala, atau mengirim data secara teratur ke server.

Namun, perlu diingat bahwa penggunaan thread programming dalam pemrograman jaringan juga memiliki beberapa tantangan, seperti sinkronisasi antar thread dan manajemen sumber daya yang kompleks. Oleh karena itu, perlu mempertimbangkan kebutuhan program dan tingkat keahlian para pengembang dalam menggunakan thread programming sebelum mengimplementasikannya dalam aplikasi jaringan.

Java menyediakan dukungan yang kuat untuk thread programming dengan menyediakan kelas-kelas dan metode-metode untuk membuat dan mengendalikan thread. Berikut adalah beberapa hal yang perlu diketahui dalam thread programming di Java:

1. Membuat thread: untuk membuat thread baru di Java, kita dapat membuat subclass dari kelas Thread dan meng-override metode run(). Metode run() akan berisi kode yang akan dijalankan di dalam thread. Selain itu, kita juga dapat membuat thread dengan mengimplementasikan interface Runnable dan meneruskannya ke kelas Thread.
2. Menjalankan thread: untuk menjalankan thread, kita dapat memanggil metode start() pada objek Thread. Metode start() akan membuat thread baru dan memanggil metode run() di dalam thread tersebut.
3. Scheduling thread: Java menyediakan mekanisme yang kuat untuk mengatur scheduling thread dengan menggunakan metode sleep() dan yield() pada objek Thread. Metode sleep() akan menghentikan thread selama beberapa waktu, sedangkan metode yield() akan memberi kesempatan kepada thread lain untuk dieksekusi.
4. Sinkronisasi thread: Java juga menyediakan beberapa mekanisme untuk sinkronisasi antara thread, seperti lock dan condition, yang dapat digunakan untuk menghindari masalah seperti race condition dan deadlock.
5. Komunikasi thread: untuk melakukan komunikasi antara thread, kita dapat menggunakan objek shared memory atau mekanisme messaging seperti Java Messaging Service (JMS).

Selain itu, Java juga menyediakan beberapa kelas dan antarmuka lainnya untuk mengatur thread, seperti Executor, Callable, dan Future. Dengan menggunakan fitur-fitur ini, pengembang dapat memaksimalkan kinerja aplikasi dan menghindari masalah-masalah yang mungkin timbul dalam thread programming.

Untuk melakukan thread network programming di Java, kita dapat menggunakan beberapa kelas yang disediakan oleh Java, seperti:

1. java.net.Socket: kelas ini digunakan untuk membuat koneksi jaringan dan melakukan komunikasi dengan server. Untuk melakukan operasi I/O pada koneksi jaringan, kita dapat menggunakan kelas InputStream dan OutputStream.
2. java.net.ServerSocket: kelas ini digunakan untuk membuat server socket dan mendengarkan koneksi masuk dari klien. Ketika koneksi masuk diterima, server dapat membuat thread baru untuk menangani koneksi tersebut.
3. java.util.concurrent.ExecutorService: kelas ini digunakan untuk mengelola pool thread dan mengeksekusi tugas secara asynchronous. Dengan menggunakan ExecutorService, kita dapat membuat banyak thread secara bersamaan dan mengoptimalkan penggunaan sumber daya CPU.
4. java.util.concurrent.Future: kelas ini digunakan untuk mewakili hasil dari tugas yang sedang berjalan di thread. Dengan menggunakan Future, kita dapat melakukan operasi asynchronous dan memantau kemajuan dari tugas yang sedang berjalan.
5. java.util.concurrent.ConcurrentHashMap: kelas ini digunakan untuk mengelola data yang dapat diakses secara bersamaan oleh beberapa thread. ConcurrentHashMap menawarkan performa yang baik dan keamanan data yang lebih baik daripada HashMap.
6. java.util.concurrent.locks.Lock: kelas ini digunakan untuk mengatur akses ke data yang dibagi oleh beberapa thread. Lock menawarkan fitur yang lebih fleksibel dan aman daripada synchronized block.

Dengan menggunakan kelas-kelas tersebut, kita dapat melakukan thread network programming di Java dan mengoptimalkan kinerja aplikasi yang berhubungan dengan jaringan.

Berikut adalah contoh program Java yang menggunakan thread untuk melakukan koneksi jaringan dengan server:

Contoh 1 :

```
 1. import java.net.*;
 2. import java.io.*;
 3.
 4. public class Client implements Runnable {
 5.
 6.     private String serverName;
 7.     private int serverPort;
 8.
 9.     public Client(String serverName, int serverPort) {
10.         this.serverName = serverName;
11.         this.serverPort = serverPort;
12.     }
13.
14.     public void run() {
15.         try {
16.             System.out.println("Connecting to " + serverName + " on port " + serverPort);
17.             Socket clientSocket = new Socket(serverName, serverPort);
18.             System.out.println("Connected to " + clientSocket.getRemoteSocketAddress());
19.
20.             BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
21.             String message = in.readLine();
22.             System.out.println("Server says: " + message);
23.
24.             clientSocket.close();
25.         } catch (IOException e) {
26.             e.printStackTrace();
27.         }
```

```
28.     }
29.
30.     public static void main(String[] args) {
31.         String serverName = "localhost";
32.         int serverPort = 8080;
33.         Thread clientThread = new Thread(new Client(serverName, serverPort));
34.         clientThread.start();
35.     }
36. }
37.
```

Dalam contoh program di atas, kelas Client mengimplementasikan interface Runnable, yang memungkinkan objek kelas Client untuk dijalankan di dalam thread. Ketika program dijalankan, objek kelas Client akan dibuat dan dijalankan di dalam thread baru menggunakan metode start(). Dalam metode run(), kelas Client membuat koneksi ke server menggunakan kelas Socket, dan membaca pesan dari server menggunakan kelas BufferedReader. Setelah pesan diterima, koneksi ditutup menggunakan metode close().

Dalam metode main(), program membuat thread baru menggunakan objek kelas Thread dan memasukkan objek kelas Client ke dalam thread tersebut menggunakan constructor Thread(Runnable). Kemudian, thread baru dijalankan menggunakan metode start(). Dengan menggunakan thread, program di atas dapat menjalankan operasi jaringan secara asynchronous dan mengoptimalkan kinerja aplikasi yang berhubungan dengan jaringan.

Berikut adalah contoh program Java yang menggunakan thread untuk membuat server socket dan menangani koneksi masuk dari klien:

Contoh 2 :

```
import java.net.*;
import java.io.*;

public class Server implements Runnable {

    private int serverPort;

    public Server(int serverPort) {
        this.serverPort = serverPort;
    }

    public void run() {
        try {
            System.out.println("Starting server on port " + serverPort);
            ServerSocket serverSocket = new ServerSocket(serverPort);

            while (true) {
                System.out.println("Waiting for client connection...");
                Socket clientSocket = serverSocket.accept();
                System.out.println("Client connected from " +
clientSocket.getRemoteSocketAddress());

                Thread clientThread = new Thread(new ClientHandler(clientSocket));
                clientThread.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        int serverPort = 8080;
```

```
        Thread serverThread = new Thread(new Server(serverPort));
        serverThread.start();
    }
}

class ClientHandler implements Runnable {

    private Socket clientSocket;

    public ClientHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    public void run() {
        try {
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            out.println("Hello, you are connected to the server!");

            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Dalam contoh program di atas, kelas Server dan ClientHandler mengimplementasikan interface Runnable, yang memungkinkan objek kelas Server dan ClientHandler untuk dijalankan di dalam thread. Ketika program dijalankan, objek kelas Server akan dibuat dan dijalankan di dalam thread baru menggunakan metode start(). Dalam metode run() kelas Server membuat server socket menggunakan kelas ServerSocket dan menunggu koneksi masuk dari klien menggunakan metode accept(). Ketika koneksi masuk diterima, kelas Server membuat thread baru menggunakan objek kelas Thread dan memasukkan objek kelas ClientHandler ke dalam thread tersebut menggunakan constructor Thread(Runnable). Kemudian, thread baru dijalankan menggunakan metode start().

Dalam kelas ClientHandler, metode run() membaca pesan dari klien menggunakan kelas PrintWriter dan menutup koneksi menggunakan metode close(). Dengan menggunakan thread, program di atas dapat menangani banyak koneksi dari klien secara bersamaan dan mengoptimalkan kinerja aplikasi server yang berhubungan dengan jaringan. Berikut adalah cara untuk menjalankan program Server dan Client yang telah dijelaskan sebelumnya:

Compile program Server dan Client menggunakan perintah berikut pada command prompt atau terminal:

```
javac Server.java
javac Client.java
```

Jalankan program Server menggunakan perintah berikut pada command prompt atau terminal:

```
java Server
```

Program Server akan menampilkan pesan "Starting server on port 8080" dan menunggu koneksi masuk dari klien. Jalankan program Client menggunakan perintah berikut pada command prompt atau terminal:

```
java Client
```

Program Client akan membuat koneksi ke server yang berjalan pada port 8080 dan menampilkan pesan "Server says: Hello, you are connected to the server!".

Setelah selesai, tekan Ctrl+C pada command prompt atau terminal untuk menghentikan program Server dan Client.

Catatan: Pastikan program Server dijalankan terlebih dahulu sebelum program Client, dan pastikan port yang digunakan oleh Server tidak sedang digunakan oleh program lain. Jika port yang digunakan sedang digunakan oleh program lain, program Server akan gagal dijalankan.

Berikut adalah contoh program Java untuk sisi server dan client yang menggunakan thread untuk menangani koneksi jaringan:

Contoh 3 :

Program Server:

```
 1. import java.io.*;
 2. import java.net.*;
 3.
 4. public class Server {
 5.     public static void main(String[] args) {
 6.         try {
 7.             ServerSocket serverSocket = new ServerSocket(8080);
 8.             System.out.println("Server started on port 8080...");
 9.
10.             while (true) {
11.                 Socket clientSocket = serverSocket.accept();
12.                 System.out.println("New client connected from " +
clientSocket.getRemoteSocketAddress());
13.
14.                 ClientHandler clientHandler = new ClientHandler(clientSocket);
15.                 Thread clientThread = new Thread(clientHandler);
16.                 clientThread.start();
17.             }
18.         } catch (IOException e) {
19.             e.printStackTrace();
20.         }
21.     }
22. }
23.
24. class ClientHandler implements Runnable {
25.     private Socket clientSocket;
26.
27.     public ClientHandler(Socket clientSocket) {
28.         this.clientSocket = clientSocket;
29.     }
30.
31.     @Override
32.     public void run() {
33.         try {
34.             BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
35.             PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
36.
37.             String inputLine;
38.             while ((inputLine = in.readLine()) != null) {
39.                 System.out.println("Received message from " +
clientSocket.getRemoteSocketAddress() + ": " + inputLine);
40.                 out.println("Server received message: " + inputLine);
41.             }
42.
43.             System.out.println("Client disconnected: " +
clientSocket.getRemoteSocketAddress());
```

```
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Program Client:

```
import java.io.*;
import java.net.*;

public class Client {
    public static void main(String[] args) {
        try {
            Socket clientSocket = new Socket("localhost", 8080);

            BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

            out.println("Hello, server!");

            String inputLine = in.readLine();
            System.out.println("Server says: " + inputLine);

            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Dalam program Server di atas, setiap kali klien terhubung ke server, server akan membuat thread baru untuk menangani koneksi klien tersebut. Kelas ClientHandler dijalankan dalam thread tersebut dan membaca masukan dari klien menggunakan kelas BufferedReader, kemudian menulis balasan ke klien menggunakan kelas PrintWriter. Kelas ClientHandler akan terus membaca masukan dari klien sampai klien memutuskan koneksi.

Dalam program Client di atas, klien membuat koneksi ke server menggunakan kelas Socket dan menulis pesan ke server menggunakan kelas PrintWriter. Kemudian, klien membaca balasan dari server menggunakan kelas BufferedReader. Setelah menerima balasan dari server, klien menampilkan pesan tersebut ke layar dan menutup koneksi.

Komunikasi antar thread dalam aplikasi jaringan

Komunikasi antar thread dalam aplikasi jaringan penting untuk menjaga sinkronisasi antara koneksi jaringan yang berbeda. Dalam aplikasi jaringan, terdapat beberapa teknik komunikasi antar thread yang dapat digunakan, di antaranya adalah:

1. Shared variables: teknik ini melibatkan penggunaan variabel yang dapat diakses oleh semua thread. Thread dapat membaca dan menulis ke variabel tersebut, dan dengan demikian dapat berkomunikasi antara satu sama lain. Namun, teknik ini juga dapat menimbulkan masalah sinkronisasi antara thread jika variabel tersebut diakses secara bersamaan.

2. Message passing: teknik ini melibatkan pengiriman pesan antar thread melalui mekanisme tertentu, seperti antrean pesan atau soket. Thread pengirim akan mengirim pesan ke thread penerima, dan thread penerima akan menerima pesan tersebut dan memprosesnya sesuai dengan tujuan aplikasi.
3. Callbacks: teknik ini melibatkan pemanggilan fungsi yang ditentukan oleh thread lain. Thread yang menerima panggilan tersebut akan menjalankan fungsi dan mengirim kembali hasilnya ke thread pemanggil. Teknik ini sering digunakan dalam aplikasi jaringan yang asinkronus, di mana thread klien akan memanggil fungsi pada thread server dan melanjutkan eksekusi tanpa harus menunggu hasil kembali.
4. Synchronization primitives: teknik ini melibatkan penggunaan alat sinkronisasi seperti mutex, semaphore, dan monitor untuk mengontrol akses ke sumber daya bersama. Thread yang ingin mengakses sumber daya bersama akan memperoleh akses melalui alat sinkronisasi tersebut, dan thread lain harus menunggu sampai sumber daya tersedia sebelum dapat mengaksesnya.

Dalam aplikasi jaringan, teknik komunikasi antar thread yang paling umum digunakan adalah message passing dan callbacks. Kedua teknik tersebut memungkinkan thread untuk berkomunikasi dengan aman dan efisien tanpa perlu membagi variabel atau mengakses sumber daya bersama yang dapat menimbulkan masalah sinkronisasi.

Berikut adalah contoh penggunaan teknik message passing dalam komunikasi antar thread dalam aplikasi jaringan menggunakan bahasa pemrograman Java:

Thread Server:

```
 1. import java.net.*;
 2. import java.io.*;
 3.
 4. public class ServerThread extends Thread {
 5.     private Socket socket = null;
 6.
 7.     public ServerThread(Socket socket) {
 8.         super("ServerThread");
 9.         this.socket = socket;
10.     }
11.
12.     public void run() {
13.         try (
14.             PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
15.             BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
16.         ) {
17.             String inputLine, outputLine;
18.             // lakukan inisialisasi objek untuk protokol tertentu
19.
20.             while ((inputLine = in.readLine()) != null) {
21.                 // proses pesan yang diterima dari klien
22.                 outputLine = processInput(inputLine);
23.                 // kirim balasan ke klien
24.                 out.println(outputLine);
25.                 if (outputLine.equals("Bye"))
26.                     break;
27.             }
28.             socket.close();
29.         } catch (IOException e) {
30.             e.printStackTrace();
31.         }
32.     }
33.
34.     private String processInput(String input) {
```

```
35.         // proses input sesuai dengan protokol tertentu
36.         return "Output";
37.     }
38. }
39.
```

Thread Client:

```
 1. import java.net.*;
 2. import java.io.*;
 3.
 4. public class ClientThread extends Thread {
 5.     private String hostName;
 6.     private int portNumber;
 7.
 8.     public ClientThread(String hostName, int portNumber) {
 9.         super("ClientThread");
10.         this.hostName = hostName;
11.         this.portNumber = portNumber;
12.     }
13.
14.     public void run() {
15.         try (
16.             Socket socket = new Socket(hostName, portNumber);
17.             PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
18.             BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
19.         ) {
20.             BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
21.             String userInput;
22.
23.             while ((userInput = stdIn.readLine()) != null) {
24.                 // kirim pesan ke server
25.                 out.println(userInput);
26.                 // baca balasan dari server
27.                 System.out.println("Server: " + in.readLine());
28.                 if (userInput.equals("Bye"))
29.                     break;
30.             }
31.         } catch (IOException e) {
32.             e.printStackTrace();
33.         }
34.     }
35. }
36.
37.
```

Pada contoh di atas, thread server dan client berkomunikasi melalui teknik message passing menggunakan soket. Thread server akan menerima koneksi dari thread client dan memulai thread baru untuk menangani koneksi tersebut. Thread client akan mengirim pesan ke thread server melalui soket, dan thread server akan memproses pesan tersebut dan mengirim balasan kembali ke thread client. Komunikasi antara thread server dan client dilakukan secara aman dan efisien tanpa perlu membagi variabel atau mengakses sumber daya bersama.

Latihan :

Berikut adalah contoh program yang menunjukkan komunikasi antar thread dalam konteks aplikasi jaringan. Program ini terdapat beberapa kesalahan logika atau penulisan script yang perlu diperbaiki sebagai latihan praktikum:

```
1. public class CommunicationDemo {
2.
3.     public static void main(String[] args) {
```

```
        MessageQueue messageQueue = new MessageQueue();

        // Thread untuk mengirim pesan
        Thread senderThread = new Thread(new MessageSender(messageQueue));
        senderThread.start();

        // Thread untuk menerima pesan
        Thread receiverThread = new Thread(new MessageReceiver(messageQueue));
        receiverThread.start();
    }
}

class MessageQueue {
    private String message;

    public synchronized void sendMessage(String message) {
        if (this.message != null) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.message = message;
        notify();
    }

    public synchronized String receiveMessage() {
        if (this.message == null) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        String receivedMessage = this.message;
        this.message = null;
        notify();
        return receivedMessage;
    }
}

class MessageSender implements Runnable {
    private MessageQueue messageQueue;

    public MessageSender(MessageQueue messageQueue) {
        this.messageQueue = messageQueue;
    }

    @Override
    public void run() {
        String[] messages = {"Hello", "How are you?", "Goodbye"};

        for (String message : messages) {
            messageQueue.sendMessage(message);
            System.out.println("Sent: " + message);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class MessageReceiver implements Runnable {
    private MessageQueue messageQueue;

    public MessageReceiver(MessageQueue messageQueue) {
```

```
73.         this.messageQueue = messageQueue;
74.     }
75.
76.     @Override
77.     public void run() {
78.         for (int i = 0; i < 3; i++) {
79.             String receivedMessage = messageQueue.receiveMessage();
80.             System.out.println("Received: " + receivedMessage);
81.         }
82.     }
83. }
84.
```

Tugas Rumah:

Multi-Threaded Network Communication

Deskripsi Tugas: Anda diminta untuk membuat program Java yang menggabungkan konsep multi-threading dengan pemrograman jaringan. Program ini harus menciptakan beberapa thread untuk meng-handle komunikasi antara klien dan server.

Spesifikasi Program:

1. **Server Program:**
    - Server harus mampu menerima koneksi dari beberapa klien secara bersamaan.
    - Setiap koneksi klien harus di-handle oleh thread terpisah.
    - Server harus dapat menerima pesan dari klien dan mem-broadcast pesan tersebut kepada semua klien yang terhubung.
2. **Klien Program:**
    - Klien harus dapat mengirimkan pesan ke server.
    - Klien harus dapat menerima pesan dari server dan menampilkannya di konsol.
    - Setiap klien harus berjalan pada thread terpisah untuk mendukung komunikasi paralel dengan server.

Catatan:

- Gunakan konsep Thread atau Executor untuk meng-handle koneksi dan komunikasi antara klien dan server.
- Server dan klien harus berkomunikasi melalui Socket.
- Pastikan untuk menangani pengecualian dan situasi error yang mungkin terjadi selama eksekusi program.
- Uji program Anda dengan membuat beberapa instance klien yang dapat terhubung dan saling berkomunikasi dengan server secara paralel.

Penting: Jangan lupa menyertakan komentar di dalam kode program untuk menjelaskan logika atau alur kerja yang diimplementasikan.