Grundlagen der Informatik für Ingenieure I

Einführung:
Parallele und Verteilte Systeme

16.1 Parallelität mit geringem Koordinierungsaufwand
16.2 Shared Data(Memory) Parallelität
16.3 Multithreading
16.4 Scheduler
16.5 verteilte Systeme/Netzwerk
16.6 Debugging paralleler Programmsysteme

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.1 Parallelität mit geringem Koordinierungsaufwand

❑ Eine Klasse von Anwendungen sind diejenigen Aufgaben, die sich leicht abspalten lassen in dem Sinne, dass **keine Notwendigkeit** besteht, **gemeinsame Daten** zu benutzen.
Das “Restsystem” ist in der Regel nur daran interessiert, eine Nachricht zu bekommen, wenn die Parallelarbeit abgeschlossen ist. Notfalls durch *Polling*, einer zyklischen Abfrage nach dem Status eines Vorgangs.

- Zu dieser beschriebenen Klasse kann man Ein-/Ausgabevorgänge zählen, die zwar parallel zu den sonstigen Aktivitäten ausgeführt werden, bei denen aber bei Lesevorgängen die “betroffenen” Daten erst nach vollständiger Verfügbarkeit weiterverarbeitet werden, bzw. bei Schreibvorgängen die auszugebenen Daten erst vollständig aufbereitete werden, bevor der Ausgabevorgang angestoßen wird.
- Häufig findet man auch in der Bildverarbeitung, z. B. beim *Raytracing,* diesen Typ von Parallelarbeit.
  Bildbereiche werden in disjunkte Teilbereiche aufgeteilt, die dann vollständig unabhängig voneinander parallel berechnet werden können. Sind alle parallelen Bearbeitungen abgeschlossen, baut ein *Masterthread* aus den Teilbildern das Gesamtbild zusammen.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.2 Shared-Data(Memory)-Parallelität

❑ Eine andere Klasse von Anwendungen haben die Eigenschaft, dass der Problemstellung eine **gemeinsame Datenbasis** zugrunde liegt. Trotzdem möchte man, mit dem Ziel der schnelleren Fertigstellung, das Problem durch mehrere Aktivitätsträger parallel bearbeiten.
Die Aufgabenstellungen sind so geartet, dass die Änderung der Daten, die von einem *Thread* bearbeitet werden, Auswirkungen auf die Berechnungen der Daten eines anderen *Threads* haben. Diese *Threads* müssen geeignet untereinander Informationen austauschen und/oder auf gemeinsame Daten überlappend, zeitversetzt, jedoch in einer **deterministischen** Weise zugreifen.

- Zu dieser Klasse von Aufgabenstellungen gehören sehr viele Lösungsverfahren in der Numerik, z. B. die Lösung von Differentialgleichungssystemen.
- E/A-Systeme machen auch häufig von dieser Möglichkeit der Parallelarbeit gebrauch, z. B. derart, dass ein *Producerthread* aufbereitete Daten in einen Puffer schreibt und ein *Consumerthread* diese Daten z. B. auf eine Datei herausschreibt. Es ist klar, dass man dafür Sorge tragen muss, dass der *Consumerthread* den Producerthread beim Zugriff auf den Puffer nicht “überholen” darf.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.2 Shared-Data(Memory)-Parallelität

❑ Das Problem bei der Bearbeitung gemeinsamer *(sharable)* Daten ist, dass der Zugriff auf Daten - selbst auf Maschinensprachenniveau - nicht atomar erfolgt; das heißt, dass der Lese- und Schreibvorgang eines Datums mehrere Zeiteinheiten dauert.

- Man kann diesen Effekt auf programmiersprachlich höherer Ebenen wie folgt demonstrieren:

```
if (mutex == 1) {
    mutex = 0;
    ...
}else {
    mutex = 1;
    .....
}
```

Wird dieser Code von mehreren *Threads* parallel ausgeführt, so besteht die Gefahr, dass der *thread_2* die Variable im *if-Statement* einmal abfragt bevor *thread_1* die Variable im Rumpf verändert hat und ein anderes mal nicht.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.2 Shared-Data(Memory)-Parallelität

- Der so deterministisch aussehende Programmcode ist in Wirklichkeit also nicht mehr deterministisch, wenn er von mehreren Threads parallel bearbeitet wird. Die tatsächliche Abarbeitungssequenz ist zufällig vom relativen Laufzeitverhalten der *Threads* zueinander abhängig.
- Es muss also dafür Sorge getragen werden, solche kritischen Abschnitte *(critical sections*) jeweils nur von einem *Thread* bearbeiten zu lassen. D.h. die Bearbeitung dieser Codeabschnitte muss **serialisiert** werden; die *Threads* werden **synchronisiert**. Man spricht auch von **Koordinierung** der *Threads.*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.2 Shared-Data(Memory)-Parallelität

- Grundlage aller Synchronisationsmechanismen ist ein Assemblerbefehl, der im Gegensatz zu allen anderen Befehlen, einen durch die Hardware sichergestellten atomaren Zugriff während eines vollständigen Lese-/ Schreibzyklus erlaubt. Sie haben architekturabhängig verschiedenen Namen, einer z. B. heißt *“read-modify-write”*.
- Der Code, der in Java die Eigenschaft hat, dass kritische Abschnitte robust gegen den Gebrauch mehrerer *Threads* sind, wird als ***thread safe*** bezeichnet. Sämtliche Klassen des Laufzeit- und Klassenbibliotheksystems haben diese Eigenschaft. Bei den von einem Benutzer erstellten Klassen, muss er selbst dafür Sorge tragen, dass das System diese Eigenschaft besitzt.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.2 Shared-Data(Memory)-Parallelität

❑ Folgende Möglichkeiten stellt Java bereit, kritische Abschnitte zu sichern:

- Der kritische Abschnitt wird im Rumpf einer Methode gekapselt

```
public synchronizid void sequentiell(...){
...
Zugriffskritischer Code
...
}
```

- Ein kritischer Abschnitt in einem Codestück wird gekapselt (die Reihenfolge wird sichergestellt!)

```
...
synchronized(this) {
    safeX = p.x();
    safeY = p.y();
}
System.out.print
            ("The point is: ("+ safeX”,”+ safeY”)”);
...
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.2 Shared-Data(Memory)-Parallelität

- Eine Variable selbst wird für einen Codeabschnitt vor Veränderung durch einen parallelen Thread geschützt:

```
...
synchronized(p) {
    safeX = p.x();
    safeY = p.y();
}
System.out.print
            ("The point is: ("+ safeX”,”+ safeY”)”);
...
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.3 Multithreading

❑ *Multithreading* haben wir aus Betriebssystemsicht bereits in Kapitel B5 diskutiert.
Grundsätzlich gibt es in Java zwei Möglichkeiten *Threads* zu erzeugen

- Ableiten von der Klasse *Thread*
- Implementieren des *Runable-Interfaces*

- Für Applets müssen wir das *Runable-Interface* implementieren(Warum?):

```
public class MeinProblem extends java.applet.Applet
implements Runnable {
...
{
```

und im Allgemeinen alle *Exceptions*, die *Thread-relevant* sind, behandeln.

- Da wir jetzt einen “eigenen” Aktivitätsträger kontrollieren müssen, benötigen wir einen *“handle”* dies zu tun. Dafür vereinbaren wir eine Variable vom Typ *Thread*:

```
Thread runner;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.3 Multithreading

❑ Methoden der Klasse *Thread* / des Interfaces *"Runnable"*

- Folgende Methoden stehen zur Kontrolle eines Threads zur Verfügung:
  - *run()-method:* diese Methode enthält den Code, den der *Thread* abarbeiten soll:

```
public void run{
...Code des Threads...
}
```

  - Die *start()-method* startet den *Thread*; im folgenden Beispiel in der Start-Methode eines Applets eingebettet. Ein "*new"* allein startet keinen Thread!:

```
public void start() {                //applet
    if (runner == null) {
    runner = new Thread(this);
    runner.start();                  //thread
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.3 Multithreading

- *stop()-method* gibt es nicht (mehr). Das Stoppen eines Thread zu einem beliebigen Zeitpunkt führt zu inkonsistenten Konstelationen.
  Lösung dieses Problems wieder an einem Applet-Beispiel:

```
public void stop() {              //applet
   if (runner != null) {
   runner = null;                 //thread;
   }
}
```

Thread terminiert mit dem *"exit"* aus der dem Thread zugeordneten *run() method.*

```
public void run() {
   Thread thisThread = Thread.currentThread();
   while (runner == thisThread) {
   try {
          Thread.sleep (someTime);
   }
   catch (interruptedException e) {}
   repaint();
   }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.3 Multithreading

- *sleep()-method:* “Schlafen legen” eines *Threads* für eine angegebene Zeit (ms) (Klassenmethode). Nach dieser Zeit wird eine *Exception* vom Typ **InterruptedException** ausgelöst.

```
try {
   Thread.sleep(delay);
} catch (InterruptedException e) {
   ....
}
```

- *yielt()-method:* Suspendiert den aufrufenden *Thread.* Der *Scheduler* entscheidet, wann dieser *Thread* wieder aktiv wird (Klassenmethode).

```
Thread.yielt();
```

- *isAlive()-method:* Gibt true zurück, falls der Thread noch nicht terminiert wurde.

```
runner.isAlive();
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.3 Multithreading

- *join()-method:* Wartet darauf, dass der angegebene Thread nicht mehr existiert. Durch Angabe einer long (Millisekunden) kann man die maximale Wartezeit beschränken.

```
annotherrunner.join(10000);
```

- currentThread()-method: Gibt die Reference auf den gerade lfd. Thread zurück (Klassenmethode).

```
Thread thisThread = Thread.currentThread()
```

- getName()-method: Gibt den Namen des Threads zurück

```
String name;
Thread runner;
name = runner.getName();
```

- setName(String name)-method: Setzt den namen des Threads neu

```
runner.setName("clock");
```

- weitere: Siehe Java-Doku; z. B. Methoden zur Prioritätensteuerung

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.3 Multithreading

- Ein vollständiges Beispiel: “Die Java-Digital-Clock”

```
import java.awt.Graphics;
import java.awt.Font;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.TimeZone;

public class DigitalClock extends java.applet.Applet
    implements Runnable {

    String name;
    Font theFont = new Font("TimesRoman",Font.BOLD,24);
    GregorianCalendar theDate;
    Thread runner;

    public void start() {
        if (runner == null) {
            runner = new Thread(this, "clock");
            runner.start();
            name = runner.getName();
            System.out.println("Thread "+name + " is running");
        }
    }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.3 Multithreading

```
public void stop() {
      if (runner != null)
          runner = null;
      }
}

public void run() {
  Thread thisThread = Thread.currentThread();
      while (runner == thisThread) {
          repaint();
          try { Thread.sleep(1000); }
          catch (InterruptedException e) { }
      }
   System.out.println("Thread "+name + " terminates now");

}
 public void paint(Graphics g)

    TimeZone tz = TimeZone.getTimeZone("CEST");

    theDate = new GregorianCalendar(tz);
    g.setFont(theFont);
    g.drawString("" + theDate.getTime(), 10, 50);
 }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.4 Scheduler (Siehe auch Kapitel B5)

- Mit *Scheduler* bezeichnet man den Teil eines Betriebssystems, der entscheidet, welche Aktivitätsträger aktiv sind. Den Entscheidungen liegen komplexe warteschlangentheoretische, stochastische und auch heuristische Verfahren zugrunde, die durch Zielfunktionen geeignet zu beeinflussen sind. So werden an einem *Scheduler* eines Echtzeitsystems andere Anforderungen gestellt, als an einem *Scheduler* eines *Time Sharing Systems*.
  - Durch das Multiplexen eines Aktivitätsträgers auf Anwenderebene, kann auch der Anwender selbst das *Threadscheduling* übernehmen. Läuft der Code auf einer Einprozessormaschine, wird er nur *“quasiparallel”* bearbeitet.
  - Was für ein Anwendersystem gilt, gilt natürlich auch für die JavaVM, die bekanntlich auch nur eine Softwareschicht oberhalb des eigentlichen Betriebssystems ist.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.4 Scheduler

- Ein Code zum Test der Eigenschaften eines Schedulers:

```
public class SchedAndThread extends Thread {
  private int delay;

    SchedAndThread(String name, float seconds) {
      super(name);
      delay = (int) (seconds * 1000);   // delays are in milliseconds
      start();                         // start up ourself!
    }
    public void run() {
      while (true) {
        System.out.println(Thread.currentThread().getName() +
                                                " delay: " + delay);
        try {
          Thread.sleep(delay);
          System.out.println(Thread.currentThread().getName() +
                                                        "waked up");
        }
        catch (InterruptedException e) {
          return;
        }
      }
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.4 Scheduler

- Ein Code zum Test der Eigenschaften eines Schedulers:

```
public class SchedAndThreadTest {

    public static void main(String argv[]) {
      new SchedAndThread("one",   1.1F);
      new SchedAndThread("two",   1.3F);
      new SchedAndThread("three", 0.5F);
      new SchedAndThread("four",  0.7F);
      new SchedAndThread("five",  0.2F);
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.4 Scheduler

```
> java SchedAndThreadTest
one delay: 1100
two delay: 1300
three delay: 500
four delay: 700
five delay: 200
five waked up
five delay: 200
five waked up
five delay: 200
three waked up
three delay: 500
five waked up
five delay: 200
four waked up
four delay: 700
five waked up
five delay: 200
three waked up
three delay: 500
five waked up
five delay: 200
one waked up
one delay: 1100
five waked up
five delay: 200
two waked up
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

❏ Kommunikation auf der Basis des *TCP/IP*-Standards, über *Sockets.* Datenströme lesen oder schreiben. Es wird unterschieden zwischen

- *(Client) Sockets* und
- *ServerSockets*
- Ein *clientside Socket* wird wie folgt instantiiert:

```
Socket connection = new Socket(hostname, portnum);
```

- Von diesem *Socket* kann nun mit *Streams* gelesen bzw. geschrieben werden:

```
DataInputStream in = DataInputStream(new
    BufferedInputStream(connection.getInputStream()));
oder
DataOutputStream out = DataOutputStream(new
    BufferedOutputStream(connection.getOutputStream()));
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

❑ Der *ServerSocket* unterscheidet sich von einem (*Client) Socket* dadurch, dass er nur einen *Socket* öffnet und darauf wartet, dass sich ein (*Client) Socket* um einen Verbindungsaufbau bemüht:

```
ServerSocket servercon = new ServerSocket(8888);
```

- Portnummern sind prinzipiell frei wählbar. Portnummern unter 1024 sind in der Regel für Systemdienste reserviert. Versucht man bereits verwendetet Portnummern zu verwenden, wird eine *Exception* ausgelöst.
- Nach dem Instantiieren des *ServerSocketObjects* muss der *Server* kundtun, dass er bereit ist, einen Verbindungsaufbau zu akzeptieren:

```
servercon.accept();
```

- An einer einfachen *Client/Server*-Konfiguration wollen wir die Dynamik einer solchen Netzwerkverbindung diskutieren. Auch hier handelt es sich um ein paralleles (verteiltes) System, dass untereinander Daten austauschen will. Im Gegensatz zur *Shared Memory* Parallelität haben die Aktivitätsträger hier disjunkte Adressräume. Informationen, die man austauschen will, müssen verschickt werden.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

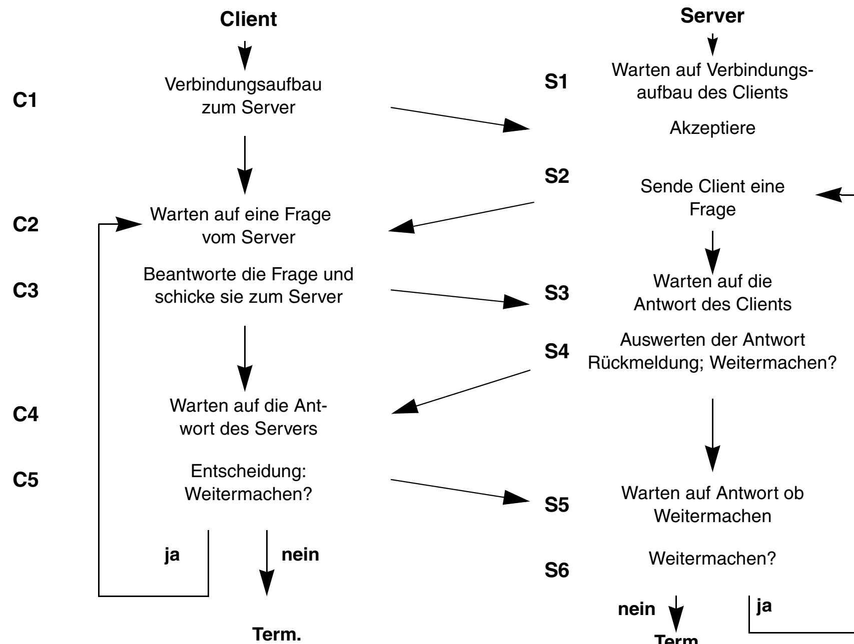
❑ Als erstes sollte man sich darüber klar werden, dass eine Kommunikation zwischen zwei (oder mehreren) Programmsystemen nur dann funktionieren kann, wenn sich diese Systeme über das verwendete **“Kommunikationsprotokoll”** einig sind. Letztendlich handelt es sich auch hierbei um eine Synchronisation der Aktivitätsträger.

- Gemäß der gesendeten oder empfangenen Daten werden sich die Programme in wohldefinierte Zustände befinden, was heißen soll, jedes Programm hat Kenntnis von dem nächsten anstehenden Kommunikationsschritt. Es ist sinnvoll, sich vorher z. B. graphisch, über das Kommunikationsprotokoll Klarheit zu verschaffen:

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

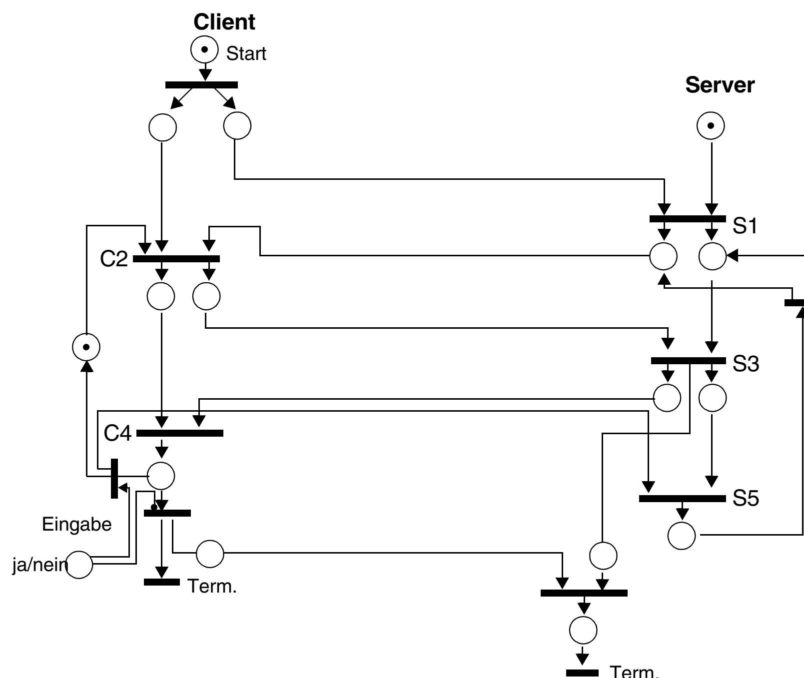
❏ Flussdiagramm und verbale Beschreibung - Problem: Korrektheit?!?



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

❏ Petrienetz - Beweisbarkeit / Übertragbarkeit



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
// Trivia Server Class
// TriviaServer.java
// Imports
import java.io.*;
import java.net.*;
import java.util.Random;

public class TriviaServer extends Thread {
  private static final int  PORTNUM = 1234;
  private static final int  WAITFORCLIENT = 0;
  private static final int  WAITFORANSWER = 1;
  private static final int  WAITFORCONFIRM = 2;
  private String[]          questions;
  private String[]          answers;
  private ServerSocket      serverSocket;
  private int               numQuestions;
  private int               num = 0;
  private int               state = WAITFORCLIENT;
  private Random            rand = new
Random(System.currentTimeMillis());
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
public TriviaServer() {
    super("TriviaServer");
    try {
      serverSocket = new ServerSocket(PORTNUM);
      System.out.println("TriviaServer up and running...");
    }
    catch (IOException e) {
     System.err.println("Exception: couldn't create socket");
      System.exit(1);
    }
  }

  public static void main(String[] args) {
    TriviaServer server = new TriviaServer();
    server.start();
  }

  public void run() {
    Socket  clientSocket = null;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
// Initialize the arrays of questions and answers
if (!initQnA()) {
 System.err.println("Error: couldn't initialize questions
                                          and answers");
  return;
}

// Look for clients and ask trivia questions
while (true) {
  // Wait for a client
  if (serverSocket == null)
    return;
  try {
    clientSocket = serverSocket.accept();
  }
  catch (IOException e) {
    System.err.println("Exception: couldn't connect to
                                      client socket");
    System.exit(1);
  }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
// Perform the question/answer processing
  try {
  DataInputStream is = new DataInputStream(new
          BufferedInputStream(clientSocket.getInputStream()));
  PrintStream os = new PrintStream(new
      BufferedOutputStream(clientSocket.getOutputStream(),false);
  String inLine, outLine;

// Output server request
  outLine = processInput(null);
  os.println(outLine);
  os.flush();
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
            // Process and output user input
            while ((inLine = is.readLine()) != null) {
              outLine = processInput(inLine);
              os.println(outLine);
              os.flush();
              if (outLine.equals("Bye."))
                break;
            }

            // Cleanup
            os.close();
            is.close();
            clientSocket.close();
          }
          catch (Exception e) {
            System.err.println("Exception: " + e);
            e.printStackTrace();
          }
        }
      }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
  private boolean initQnA() {
    try {
      File            inFile = new File("QnA.txt");
      FileInputStream inStream = new FileInputStream(inFile);
      byte[]          data = new byte[(int)inFile.length()];

// Read the questions and answers into a byte array
      if (inStream.read(data) <= 0) {
       System.err.println("Error: couldn't read qest. and answers");
        return false;
      }
// See how many question/answer pairs there are
      for (int i = 0; i < data.length; i++)
        if (data[i] == (byte)'\n')
          numQuestions++;
      numQuestions /= 2;
      questions = new String[numQuestions];
      answers = new String[numQuestions];

// Parse the questions and answers into arrays of strings
      int start = 0, index = 0;
      boolean isQ = true;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
for (int i = 0; i < data.length; i++)
      if (data[i] == (byte)'\n') {
        if (isQ) {
          questions[index] = new String(data,0, start, i - start - 1);
          isQ = false;
        }
        else {
          answers[index] = new String(data, 0, start, i - start - 1);
          isQ = true;
          index++;
        }
        start = i + 1;
      }
  }
  catch (FileNotFoundException e) {
    System.err.println("Exception:couldn't find the fortune file");
    return false;
  }
  catch (IOException e) {
    System.err.println("Except.:I/O error trying to read quest.");
    return false;
  }
 return true;
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
String processInput(String inStr) {
  String outStr = null;

  switch (state) {
  case WAITFORCLIENT:
    // Ask a question
    outStr = questions[num];
    state = WAITFORANSWER;
    break;

  case WAITFORANSWER:
    // Check the answer
    if (inStr.equalsIgnoreCase(answers[num]))
      outStr = "That's correct! Want another? (y/n)";
    else
      outStr = "Wrong, the correct answer is " + answers[num] +
                                 ". Want another? (y/n)";
    state = WAITFORCONFIRM;
    break;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
        case WAITFORCONFIRM:
          // See if they want another question
          if (inStr.equalsIgnoreCase("y")) {
            num = Math.abs(rand.nextInt()) % questions.length;
            outStr = questions[num];
            state = WAITFORANSWER;
          }
          else {
            outStr = "Bye.";
            state = WAITFORCLIENT;
          }
          break;
        }
        return outStr;
      }
    }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
// Trivia Class
// Trivia.java
// Imports
import java.io.*;
import java.net.*;
public class Trivia {
  private static final int PORTNUM = 1234;

  public static void main(String[] args) {
    Socket          socket = null;
    DataInputStream in = null;
    PrintStream     out = null;
    String          address;
    // Check the command-line args for the host address
    if (args.length != 1) {
      System.out.println("Usage: java Trivia <address>");
      return;
    }
    else
      address = args[0];
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
        // Initialize the socket and streams
        try {
          socket = new Socket(address, PORTNUM);
          in = new DataInputStream(socket.getInputStream());
          out = new PrintStream(socket.getOutputStream());
        }
        catch (IOException e) {
         System.err.println("Exception: couldn't create stream socket");
          System.exit(1);
        }
        // Process user input and server responses
        try {
          StringBuffer  str = new StringBuffer(128);
          String        inStr = null;
          int           c;
          while ((inStr = in.readLine()) != null) {
            System.out.println("Server: " + inStr);
            if (inStr.equals("Bye."))
              break;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
    while ((c = System.in.read()) != '\n')
              str.append((char)c);
            System.out.println("Client: " + str);
            out.println(str.toString());
            out.flush();
            str.setLength(0);
          }

          // Cleanup
          out.close();
          in.close();
          socket.close();
        }
        catch (IOException e) {
          System.err.println("Exception: I/O error trying to talk to
                                                          server");
        }
      }
    }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.5 Verteilte Systeme/Netzwerk

```
faui40c: > java TriviaServer
TriviaServer up and running...


faui02b:> java Trivia faui40c.informatik.uni-erlangen.de

Server: What caused the craters on the moon?
meteorites
Client: meteorites
Server: That's correct! Want another? (y/n)
y
Client: y
Server: How old is the Earth (in millions of years)?
12000
Client: 12000
Server: Wrong, the correct answer is 4600. Want another? (y/n)
y
Client: y
Server: Is the Galaxy rotating?
yes
Client: yes
Server: That's correct! Want another? (y/n)
y
Client: y
Server: How old is the Earth (in millions of years)?
4600
Client: 4600
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.


```
Server: That's correct! Want another? (y/n)
y
Client: y
Server: What is the name for the band of dim light encircling the sky?
corona
Client: corona
Server: Wrong, the correct answer is Milky Way. Want another? (y/n)
y
Client: y
Server: How far away is the sun (in millions of miles)?
93
Client: 93
Server: That's correct! Want another? (y/n)
n
Client: n
Server: Bye.
>
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

16.6 Debugging Paralleler Programmsysteme

- ❑ Das Eingrenzen von Laufzeitfehlern ist bei parallelen Programmsystemen ein ausgesprochen schwieriges Unterfangen, da - wie immer man an eine Fehlersuche herangeht - eingestreuter Code notwendig ist, der das dynamischen Verhalten des Systems verändert. Noch schwerwiegender ist, dass man in der Lage sein muss, zu bestimmten Zeitpunkten *“Snapshots”*, in Form von Datensicherungen, durchzuführen. Diese Maßnahmen sind zwangsläufig synchronisierend - verändern den dynamischen Ablauf also erheblich.
- ❑ Da diese “haarigen” nicht deterministischen Laufzeitfehler - in der Regel Koordinierungsfehler bzw. Kommunikationsprotokollfehler sind, gibt es nur einen Weg:
  Den “gedanklichen” Strukturvergleich zwischen Protokoll und Implementierung.
- ❑ Es gibt sogenannte “Paralleldebugger”, ihre Eignung beschränkt sich aber auf SPMD *(single programm multiple data)* Programmsysteme; für numerische Lösungen allerdings ein wichtiges Programmiermodell.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.