Script **generated by TTT**

Title: Grundlagen_Betriebssysteme (21.12.2015)

Date: Mon Dec 21 13:45:14 CET 2015

Duration: 84:59 min

Pages: 28

Prozesskommunikation

Disjunkte Prozesse, d.h. Prozesse, die völlig isoliert voneinander ablaufen, stellen eher die Ausnahme dar. Häufig finden Wechselwirkungen zwischen den Prozessen statt ⇒ Prozesse interagieren. Die Unterstützung der Prozessinteraktion stellt einen unverzichtbaren Dienst dar.

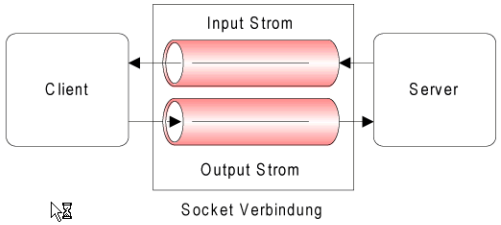
Fragestellungen

Dieser Abschnitt beschäftigt sich mit den Mechanismen von Rechensystemen zum Austausch von Informationen zwischen Prozessen.

- Kommunikationsarten.
- nachrichtenbasierte Kommunikation, insbesondere Client-Server-Modell.
- Netzwerkprogrammierung auf der Basis von Ports und Sockets.

Einführung

Ein ***Socket*** definiert einen einfachen, bidirektionalen Kommunikationskanal zwischen 2 Rechensystemen, mit Hilfe dessen 2 Prozesse über ein Netz miteinander kommunizieren können.



Socket Grundlagen

Socket Grundlagen

Sockets abstrahieren von den technischen Details eines Netzes, z.B. Übertragungsmedium, Paketgröße, Paketwiederholung bei Fehlern, Netzadressen.

Ein Socket kombiniert 2 Ströme, einen Input- und einen Output-Strom.

Ein Socket unterstützt die folgenden Basisoperationen:

- richte Verbindung zu entferntem Rechner ein ("connect").
- sende Daten.
- empfange Daten.
- schließe Verbindung.
- assoziiere Socket mit einem Port.
- warte auf eintreffende Daten ("listen").
- akzeptiere Verbindungswünsche von entfernten Rechnern (bzgl. assoziiertem Port).

Server Protokoll

Ein Server kommuniziert mit einer Menge von Clients, die a priori nicht bekannt sind. Ein Server benötigt eine Komponente (z.B. ein Verteiler-Thread), die auf eintreffende Verbindungswünsche reagiert.

Informeller Ablauf aus Serversicht

1. Erzeugen eines SocketServer und Binden an einen bestimmten Port.
2. Warten auf Verbindungswünsche von Clients.
3. Austausch von Daten zwischen Client und Server entsprechend einem wohldefinierten Protokoll (z.B. HTTP).
4. Schließen einer Verbindung (durch Server, durch Client oder durch beide); weiter bei Schritt 2.

Programmstück

```
Socket socket; //reference to socket
ServerSocket port; //the port the server listens to
try {
     port = new ServerSocket(10001, ...);
     socket = port.accept(); //wait for client call
     // communicate with client
     socket.close();
}
catch (IOException e) {e.printStackTrace();}
```

Für das Abhören des Ports kann ein eigener Verteiler-Thread spezifiziert werden; die Bearbeitung übernehmen sogenannte Worker-Threads.

Client Protokoll

Der Client initiiert eine Socket-Verbindung durch Senden eines Verbindungswunsches an den Port des Servers.

Informeller Ablauf aus Clientsicht

1. Erzeugen einer Socket Verbindung.
2. Austausch von Daten zwischen Client und Server über die Duplex-Verbindung entsprechend einem wohldefinierten Protokoll (z.B. HTTP).
3. Schließen einer Verbindung (durch Server, durch Client oder durch beide).

Programmstück

```
Socket connection; //reference to socket
try {
     connection = new Socket("www11.in.tum.de", 10001);
     ........ // communicate with client
     connection.close();
}
catch (IOException e) {e.printStackTrace();}
```


Bidirektionale Stromverbindung

Sockets bestehen aus 2 Strömen für die Duplexverbindung zwischen Client und Server.

Schreiben auf Socket

```
void writeToSocket(Socket sock, String str) throws IOException {
     oStream = sock.getOutputStream();
     for (int k = 0; k < str.length(); k++)
         oStream.write(str.charAt(k));
}
```

Lesen von Socket

```
String readFromSocket(Socket sock) throws IOException {
     iStream = sock.getInputStream();
     String str = "";
     char c;
     while ( (c = (char) iStream.read()) != '\n')
         str = str + c;
     return str;
}
```


Java Socket Class

Java unterstützt die beiden grundlegenden Klassen:

`java.net.Socket` zur Realisierung der Client-Seite einer Socket.

`java.net.ServerSocket` zur Realisierung der Server-Seite einer Socket.

Client-Seite einer Socket
Server-Seite einer Socket

Client-Seite einer Socket

Constructor

```
public Socket(String host, int port)
    throws UnknownHostException, IOException
```

Der Parameter host ist ein Rechnername, z.B. www11.in.tum.de.

Information über eine Socket

```
public InetAddress getInetAddress();
```

liefert als Ergebnis den Namen und IP-Adresse des entfernten Rechners, zu dem die Socket-Verbindung existiert.

```
public int getPort();
```

liefert als Ergebnis die Nummer des Ports, mit dem die Socket-Verbindung am entfernten Rechner assoziiert ist.

```
public int getLocalPort();
```

liefert als Ergebnis die Nummer des Ports, mit dem die Socket-Verbindung am lokalen Rechner assoziiert ist.

Ein-/Ausgabe

```
public InputStream getInputStream() throws IOException;
```

liefert den InputStream, von dem Daten gelesen werden können.

```
public OutputStream getOutputStream() throws IOException;
```

liefert den OutputStream, in dem Daten geschrieben werden können.

Java Socket Class

Java unterstützt die beiden grundlegenden Klassen:

`java.net.Socket` zur Realisierung der Client-Seite einer Socket.

`java.net.ServerSocket` zur Realisierung der Server-Seite einer Socket.

Client-Seite einer Socket
Server-Seite einer Socket

Beispiel - Generische Client/Server Klassen

Das Beispiel zeigt eine allgemeine Client/Server-Anwendung, wobei der Ausgangspunkt eine generische ClientServer Klasse ist, aus der konkrete Services abgeleitet werden können.

ClientServer Klasse
EchoServer Klasse
EchoClient Klasse

EchoServer Klasse

Der EchoServer sendet den String einer Client Anfrage wieder zurück.

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

public class EchoServer extends ClientServer {
    private ServerSocket port;
    private Socket socket;
    public EchoServer (int portNum, int nBackLog) {
        try { port = new ServerSocket(portNum, nBackLog);
        } catch (IOException e) { e.printStackTrace(); }
    }
    public void run() {
        try {
            System.out.println("Echo server at "
                + InetAddress.getLocalHost() + " waiting for connections ");
                while (true) {
                    socket = port.accept();
                    System.out.println("Accepted a connection from " + socket.getInetAddress
                    () );
                    provideService(socket);
                    socket.close();
                    System.out.println("Closed the connection\n");
                }
        } catch (IOException e) { e.printStackTrace(); }
    }
    protected void provideService (Socket socket) {
```

```
public class EchoServer extends ClientServer {
    private ServerSocket port;
    private Socket socket;
    public EchoServer (int portNum, int nBackLog) {
        try { port = new ServerSocket(portNum, nBackLog);
        } catch (IOException e) { e.printStackTrace(); }
    }
    public void run() {
        try {
            System.out.println("Echo server at "
                + InetAddress.getLocalHost() + " waiting for connections ");
                while (true) {
                    socket = port.accept();
                    System.out.println("Accepted a connection from " + socket.getInetAddress
                    () );
                    provideService(socket);
                    socket.close();
                    System.out.println("Closed the connection\n");
                }
        } catch (IOException e) { e.printStackTrace(); }
    }
    protected void provideService (Socket socket) {
        String str = "";
        try {
            writeToSocket(socket, "Hello, how may I help you ?\n");
            do {
```



```
                    provideService(socket);
                    socket.close();
                    System.out.println("Closed the connection\n");
                }
        } catch (IOException e) { e.printStackTrace(); }
    }
    protected void provideService (Socket socket) {
        String str = "";
        try {
            writeToSocket(socket, "Hello, how may I help you ?\n");
            do {
                str = readFromSocket(socket);
                if (str.toLowerCase().equals("goodby"))
                    writeToSocket(socket, "Goodbye\"n);
                else
                    writeToSocket(socket, "You said '" + str + "'\n");
                } while (!str.toLowerCase().equals("goodbye") );
            } catch (IOException e) { e.printStackTrace(); }
    }
    public static void main (String args[]) {
        EchoServer server = new EchoServer(10001, 3);
        server.start();
    }
}
```



```
    private Socket socket;
    public EchoServer (int portNum, int nBackLog) {
        try { port = new ServerSocket(portNum, nBackLog);
        } catch (IOException e) { e.printStackTrace(); }
    }
    public void run() {
        try {
            System.out.println("Echo server at "
                + InetAddress.getLocalHost() + " waiting for connections ");
                while (true) {
                    socket = port.accept();
                    System.out.println("Accepted a connection from " + socket.getInetAddress
                    () );
                    provideService(socket);
                    socket.close();
                    System.out.println("Closed the connection\n");
                }
        } catch (IOException e) { e.printStackTrace(); }
    }
    protected void provideService (Socket socket) {
        String str = "";
        try {
            writeToSocket(socket, "Hello, how may I help you ?\n");
            do {
                str = readFromSocket(socket);
```



```
                while (true) {
                    socket = port.accept();
                    System.out.println("Accepted a connection from " + socket.getInetAddress
                    () );
                    provideService(socket);
                    socket.close();
                    System.out.println("Closed the connection\n");
                }
        } catch (IOException e) { e.printStackTrace(); }
    }
    protected void provideService (Socket socket) {
        String str = "";
        try {
            writeToSocket(socket, "Hello, how may I help you ?\n");
            do {
                str = readFromSocket(socket);
                if (str.toLowerCase().equals("goodby"))
                    writeToSocket(socket, "Goodbye\"n);
                else
                    writeToSocket(socket, "You said '" + str + "'\n");
                } while (!str.toLowerCase().equals("goodbye") );
            } catch (IOException e) { e.printStackTrace(); }
    }
    public static void main (String args[]) {
        EchoServer server = new EchoServer(10001, 3);
```

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

public class EchoClient extends ClientServer {
      protected Socket socket;
      public EchoClient (String url, int port) {
          try { port = new Socket(url, port);
              System.out.println("Client: connected to " + url + ":" + port);
          } catch (IOException e) { e.printStackTrace(); System.exit(1); }
      }
      public void run() {
          try {
              requestService(socket);
              socket.close();
              System.out.println("Client: connection closed");
          } catch (IOException e) { System.out.println(e.getMessage()); e.printStackTrace(); }
      }
      protected void requestService (Socket socket) throws IOException {
          String servStr = readFromSocket(socket);
          System.out.println("Server: " + servStr);
          System.out.println("Client: type a line or 'goodbye' to quit");
          if (servStr.substring(0, 5).equals("Hello")) {
              String userStr ="";
              do {
                  userStr = readFromKeyboard();
```



```
          } catch (IOException e) { e.printStackTrace(); System.exit(1); }
      }
      public void run() {
          try {
              requestService(socket);
              socket.close();
              System.out.println("Client: connection closed");
          } catch (IOException e) { System.out.println(e.getMessage()); e.printStackTrace(); }
      }
      protected void requestService (Socket socket) throws IOException {
          String servStr = readFromSocket(socket);
          System.out.println("Server: " + servStr);
          System.out.println("Client: type a line or 'goodbye' to quit");
          if (servStr.substring(0, 5).equals("Hello")) {
              String userStr ="";
              do {
                  userStr = readFromKeyboard();
                  writeToSocket(socket, userStr + "\n");
                  servStr = readFromSocket(socket);
                  System.out.println("Server: " + servStr);
              } while (!userStr.toLowerCase().equals("goodbye"));
          }
      }
      protected String readFromKeyboard() throws IOException {
          BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
```



```
      protected Socket socket;
      public EchoClient (String url, int port) {
          try { port = new Socket(url, port);
              System.out.println("Client: connected to " + url + ":" + port);
          } catch (IOException e) { e.printStackTrace(); System.exit(1); }
      }
      public void run() {
          try {
              requestService(socket);
              socket.close();
              System.out.println("Client: connection closed");
          } catch (IOException e) { System.out.println(e.getMessage()); e.printStackTrace(); }
      }
      protected void requestService (Socket socket) throws IOException {
          String servStr = readFromSocket(socket);
          System.out.println("Server: " + servStr);
          System.out.println("Client: type a line or 'goodbye' to quit");
          if (servStr.substring(0, 5).equals("Hello")) {
              String userStr ="";
              do {
                  userStr = readFromKeyboard();
                  writeToSocket(socket, userStr + "\n");
                  servStr = readFromSocket(socket);
                  System.out.println("Server: " + servStr);
              } while (!userStr.toLowerCase().equals("goodbye"));
```

"Hello how may I help you"

lesen Nutzereingabe

sende an Server

lies Antwort vom Server

Disjunkte Prozesse, d.h. Prozesse, die völlig isoliert voneinander ablaufen, stellen eher die Ausnahme dar. Häufig finden Wechselwirkungen zwischen den Prozessen statt ⇒ Prozesse interagieren. Die Unterstützung der Prozessinteraktion stellt einen unverzichtbaren Dienst dar.

Fragestellungen

Dieser Abschnitt beschäftigt sich mit den Mechanismen von Rechensystemen zum Austausch von Informationen zwischen Prozessen.

- Kommunikationsarten.
- nachrichtenbasierte Kommunikation, insbesondere Client-Server-Modell.
- Netzwerkprogrammierung auf der Basis von Ports und Sockets.

Dateisysteme

Zentrale Aufgabe des Dateisystems ist es die besonderen Eigenschaften externer Speichermedien optimal umzusetzen und Applikationen einen effizienten Zugriff auf die persistent gespeicherten Daten zu ermöglichen. Es gelten folgende grundlegende Forderungen

a) Speicherung großer Informationsmengen (Video)

b) kein Datenverlust auch bei Prozess- / Systemabsturz

c) nebenläufiger Zugriff durch mehrere Prozesse

Fragestellungen

Dieser Abschnitt beschäftigt sich mit den Mechanismen eines Rechensystems zur dauerhaften (persistenten) Speicherung von Programmen und Daten:

Charakteristika von Dateisystemen.

Schichtenmodell eines Dateisystems.

Charakteristika von Dateisystemen

Jedes Dateisystem unterstützt 2 grundlegende Abstraktionen:

Datei : Behälter für die persistente Speicherung jeglicher Information.

Verzeichnisse : spezielle Dateien zur Strukturierung externer Speichermedien.

blockorientierter Datentransfer zwischen externem Speicher und Arbeitsspeicher.

Typische Blockgrößen: 512 Byte - 4 KByte.

Charakteristika der Dateinutzung (empirisch ermittelt):

a) Dateien sind meist klein (wenige KBytes).

b) Dateien werden häufiger gelesen, seltener geschrieben und noch seltener gelöscht.

c) vornehmlich sequentieller Zugriff.

d) Dateien werden selten von mehreren Programmen oder Personen gleichzeitig benutzt.

Für Multimedia verändert sich die Nutzungscharakteristik.

große Dateien (mehrere GByte).

gleichmäßige Zugriffsgeschwindigkeit (um Ruckeln zu vermeiden).

notwendige Übertragungsbandbreite.

Dateien

Dateien bilden in einem Dateisystem die Behälter für die dauerhafte Speicherung beliebiger Information.

in den meisten Systemen wird eine Datei als eine Folge von Bytes aufgefasst.

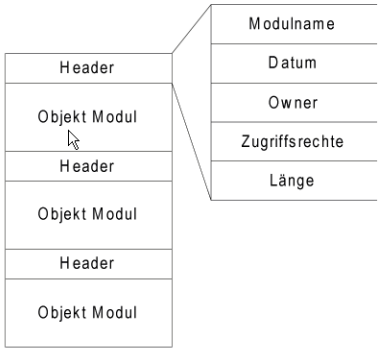
Dateinamen

in manchen Dateisystemen haben Dateinamen die Form "name.extension".

Beispiele für extension: `.c, .java, .html, .gif, .pdf, .tex, .doc, .zip, ....`

Dateiaufbau

Die interne Struktur einer Datei hängt von der jeweiligen Nutzung und Zielsetzung ab, z.B. ASCII Datei besteht aus Zeilen, die mit CR, LF abgeschlossen sind. Beispiel einer Archivdatei



Dateien

Dateien bilden in einem Dateisystem die Behälter für die dauerhafte Speicherung beliebiger Information.

in den meisten Systemen wird eine Datei als eine Folge von Bytes aufgefasst.

Dateinamen

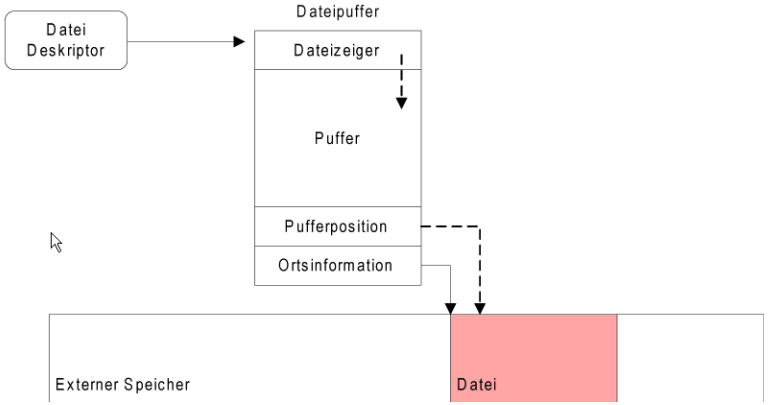
in manchen Dateisystemen haben Dateinamen die Form "name.extension".

Beispiele für extension: `.c, .java, .html, .gif, .pdf, .tex, .doc, .zip, ....`

Dateiaufbau
Operationen

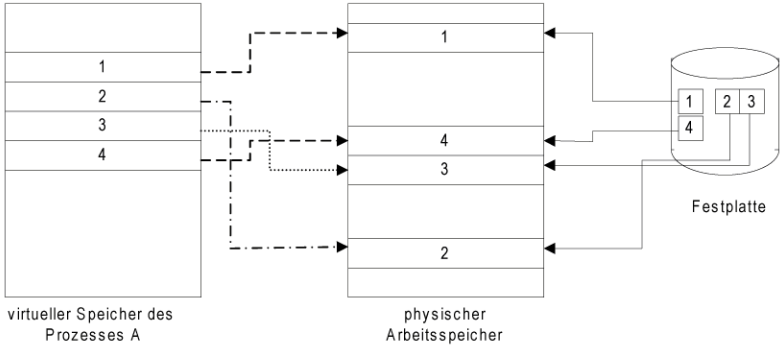
Dateipuffer

Zugriffe auf Dateien erfolgen über einen Dateideskriptor und einen Dateipuffer.



Memory-Mapped Dateien

Eine Datei oder Teile davon werden in den virtuellen Adressraum eines Prozesses eingeblendet.



Lese- und Schreiboperationen, sowohl sequentiell als auch wahlfrei, erfolgen über virtuelle Adressen.

Einblendung immer nur Vielfacher ganzer Blöcke einer Datei.

veränderte Blöcke werden meist aus Effizienzgründen zu einem späteren Zeitpunkt zurückgeschrieben.

gemeinsame Nutzung einer Datei durch mehrere Prozesse möglich.

Beispiel

Beispiel

Beispiel der Win32-Programmierschnittstelle:

```
Handle fh, fmh;
fh = CreateFile(filename, generic_read, .....);
len = GetFileSize(fh, ...);
fmh = CreateFileMapping(fh, Page_readonly, ...);
addr = MapViewOfFile(fmh, File_map_read, ...);
sprintf(addr, "Information für memory-mapped Datei");
```

Beispiel Unix Solaris

Systemaufruf mmap() spezifiziert eine Datei als memory-mapped Datei; Datei wird in den virtuellen Adressraum des Prozesses geladen.

bei Aufruf des Datei-Systemaufrufs open; Datei wird als memory-mapped Datei in den Betriebssystembereich des virtuellen Adressraums geladen. read/write-Systemaufrufe werden auf dieser memory-mapped Datei ausgeführt.

Zentrale Aufgabe des Dateisystems ist es die besonderen Eigenschaften externer Speichermedien optimal umzusetzen und Applikationen einen effizienten Zugriff auf die persistent gespeicherten Daten zu ermöglichen. Es gelten folgende grundlegende Forderungen

a) Speicherung großer Informationsmengen (Video)

b) kein Datenverlust auch bei Prozess- / Systemabsturz

c) nebenläufiger Zugriff durch mehrere Prozesse

Fragestellungen

Dieser Abschnitt beschäftigt sich mit den Mechanismen eines Rechensystems zur dauerhaften (persistenten) Speicherung von Programmen und Daten:

Charakteristika von Dateisystemen.

Schichtenmodell eines Dateisystems.