Threads und Synchronisierung (2)

Betriebssystem-Prozesse

Nicht immer wollen wir nur unsere eigenen Threads starten, sondern manchmal auch externe Prozesse. Dazu werden die (System-)Klassen **Runtime** und **Process** verwendet.

```
public class SystemExec
{
    static PrintWriter conout=new
PrintWriter(System.out,true);

static public void main(String args[])
{
    String outfile;
    String infile;
    String errfile=null;
    String command;
    Process proc=null;
    InputStreamReader out=null;
    InputStreamReader err=null;
    OutputStreamWriter in=null;
    conout.println("Programmstart\n\n");
```

```
    // Parse arguments
    if(args.length<3)
    {
        conout.println("Usage: SystemExec [-e Error-file]
Input-file Output-file Programname {Arguments}");
        System.exit(1);
    }
    int curIndex=0;
    if(args[curIndex].equals("-e") ||
       args[curIndex].equals("-E"))
    {
        curIndex++;
        errfile=args[curIndex++];
    }
    if(args.length-curIndex<3)
    {
        conout.println("Usage: SystemExec [-e Error-file]
Input-file Output-file Programname {Arguments}");
        System.exit(1);
    }
    infile=args[curIndex++];
    outfile=args[curIndex++];
    command=args[curIndex++];
    // Add up rest of arguments as args for the program
    for(int i=curIndex;i<args.length;i++)
        command+=" "+args[i];
    try
    {
        // Start program
        proc=Runtime.getRuntime().exec(command);
```

```
        // getInputStream returns an InputStream,
        // which is stdout of the command. You can read
        // the output of the program through this stream
        out=new InputStreamReader(proc.getInputStream());
        // getErrorStream returns an InputStream, which
        // is stderr of the command
        err=new InputStreamReader(proc.getErrorStream());
        // getOutputStream returns an OutputStream, which
        // is stdin of the command. What you write
        // in here is fed as input to the program
       in=new OutputStreamWriter(proc.getOutputStream());
    }
    catch(IOException e)
    {
        conout.println(e);
        System.exit(1);
    }
    // Start new Threads for handling the streams
   Thread inH=new StdinHandler(in,infile);
   Thread outH=new StdoutHandler(out,outfile);
   Thread errH=new StderrHandler(err,errfile);
   outH.start();
   errH.start();
   inH.start();
   // Wait till all are closed
   try
   {
       inH.join();
       outH.join();
       errH.join();
```

```
        proc.waitFor();
    }
    catch(InterruptedException e)
    {   conout.println(e);
    }
    // Print exit value
    conout.println("\n>>>>>The programme returned the
                exit-code "+proc.exitValue()+"<<<<<");
    conout.println("\n\nProgrammende");
    // Explicitely end Java VM
    // Needed for setting an exit value
    System.exit(0);
}
}
```

StdOut und StdErr-Handler lesen zeilenweise und schreiben in eine Datei (Source-Code auf Server)

```
public class StdinHandler extends Thread
{
     protected OutputStreamWriter stdin;
     protected String infile;

    public StdinHandler(OutputStreamWriter os,String file)
     {
         if(os==null || file==null)
             throw new IllegalArgumentException();
         stdin=os;
         infile=file;
     }
```

```
    public void run()
    {
        BufferedReader fr=null;
        try{
           fr=new BufferedReader(new FileReader(infile));
        }
        catch(IOException e) {
            // Should notify main program to stop
            // all other threads and end
            System.out.println("Stdin: "+e);
            return;
        }
        try {
            String inp=fr.readLine();
            while(inp!=null) {
                inp+="\n";
                try {
                    stdin.write(inp,0,inp.length());
                }
                catch(IOException ie) {
// We print no error here, as it may happen that the
// process has already terminated (if not all was read).
// No way to find this out (waitFor cannot be used,
// because it will block till process ended!).
// Just close down everything nicely!
                    fr.close();
                    return;
                }
                inp=fr.readLine();
            }
```

```
            // We must close stdin to signal our process,
            // that the end of the stream is reached
            try
            {
                stdin.flush();
                stdin.close();
            }
            catch(IOException e)
            {
                // See comments above!
            }
            fr.close();
        }
        catch(IOException e)
        {
            System.out.println("Stdin: "+e);
        }
    }
}
```

Producer ? Buffer ? Consumer

Ein ständig wiederkehrendes Problem ist der Austausch von Daten zwischen verschiedenen Threads. In vielen Fällen ist eine zweiseitige Kommunikation nicht nötig, kann aber problemlos aus zwei unidirektionalen Kommunikationskanälen hergestellt werden.

Folgende Punkte sollten beim Grundmuster beachtet werden:

? Sowohl der Produzent wie auch der Konsument können den Buffer übernehmen, oder dieser kann ein besonderes Objekt sein.

? Jedes Objekt sollte nur **eine** Synchronisationsaufgabe besitzen (Also nicht zugleich Buffer sein und synchronized-Methoden auch noch für andere Zwecke verwenden!).

? Denken Sie immer daran, daß u. U. der Buffer nicht mehr leer wird oder keine weiteren Werte produziert werden. In diesem Fall darf kein "ewiges" wait erfolgen (? Siehe z. B. unterbrechbare Aktivitäten)!

Hauptprogramm:

```
public class Test
{
 static PrintWriter out=new PrintWriter(System.out,true);

static public void main(String args[])
{
    out.println("Programmstart\n\n");
    URL url=null;
    Buffer buf=new Buffer(5); // Einen Buffer erzeugen
    try {
        url=new URL("http://www.fim.uni-linz.ac.at");
    }
    catch(MalformedURLException e)
    { out.println(e); System.exit(1); }
    // Producer und Consumer erzeugen
    Producer prod=new Producer(buf,url);
    Consumer cons=new Consumer(buf);
    prod.start();
    cons.start();
    try {
        prod.join();
        cons.join();
    }
    catch(InterruptedException e)
    { out.println(e); }
    out.println("\n\nProgrammende");
}
}
```

Buffer:

```
// Buffer fuer Strings;
// Laenge kann bei Erzeugung angegeben werden
public class Buffer extends Thread
{
    protected String[] buf;
    protected int last; // Naechster freier Index
    protected int EOF;
    // Wenn EOF>=0 -> Anzahl der String im Buffer bis EOF

    public Buffer(int len)
    {
        buf=new String[len];
        last=0; EOF=-1;
    }

    public synchronized void setEOF()
    {
        EOF=last;
// Wenn der Buffer leer ist, koennte der Consumer warten!
        if(EOF==0)
            notify();
    }

    public synchronized boolean isEOF()
    // Pruefen ob am Ende
    {
        return EOF==0;
    }
```

```
    public synchronized boolean put(String item)
    {
        // Einen String in den Buffer stellen
        if(EOF>=0) // Keine Eingabe nach EOF
            return false;
        // Ist ein Platz frei?
        if(last==buf.length)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
                return false;
            }
        }
        buf[last++]=item;
// ACHTUNG! Geht nur, weil nur ein Producer (sonst
// koennte damit ein anderer Producer geweckt werden)!
// Mehrere Producer -> Semaphor oder aehnliches verwenden
        notify();
        return true;
    }
```

```
    public synchronized String get()
    {
        // Einen String auslesen
        if(EOF==0)    // Nach EOF gibt es nichts mehr
            return null;
        if(last==0) // Ist der Buffer leer?
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
                return null;
            }
            if(EOF==0)
// ACHTUNG: Wenn der Consumer gewartet hat (Buffer leer),
// waehrend der Producer EOF setzte ist dies notwendig!
                return null;
        }
        String res=buf[0];
        // Array nach vorne kopieren
        System.arraycopy(buf,1,buf,0,--last);
        if(EOF>0)    // EOF anpassen wenn noetig
            EOF--;
        // ACHTUNG: Siehe Put!!!
        notify();
        return res;
    }
    }
```

Producer:

```
public class Producer extends Thread
{
    URL url;
    Buffer buf;

    public Producer(Buffer buf,URL url)
    { this.url=url; this.buf=buf; }

    public void run()
    {
        BufferedReader in=null;
        String str;
        // URL oeffnen und Daten holen
        try
        {
            URLConnection uc=url.openConnection();
            uc.setUseCaches(false);
            uc.connect();
            in=new BufferedReader(
             new InputStreamReader(uc.getInputStream()));
        }
        catch(IOException e)
        {
            System.out.println("Could not connect to"
                               +url+"\n"+e+"\n");
            return;
        }
```

```
        // LeseSchleife
        try
        {
            while((str=in.readLine())!=null)
                buf.put(str);
        }
        catch(IOException e)
        {
            System.out.println("Could not read from "
                                +url+"\n"+e+"\n");
            return;
        }
        // EOF signalisieren
        buf.setEOF();
    }
}
```

Consumer:

```
public class Consumer extends Thread
{
    Buffer buf;

    public Consumer(Buffer buf)
    {
        this.buf=buf;
    }
```

```
    public void run()
    {
        String tagBuffer="";    // Eigener Buffer wegen
       // Tags, die auch über Zeilenwechsel gehen koennen
        String line=buf.get();
        while(line!=null)
        {
            String res=new String("");
            tagBuffer+=line+"\n";
            int lt=tagBuffer.indexOf('<');
            while(lt>=0)
            { // Start eines tags
                int gt=tagBuffer.indexOf('>',lt);
                if(gt>=0)
                {
                    res+=tagBuffer.substring(0,lt);
tagBuffer=tagBuffer.substring(gt+1,tagBuffer.length());
                    lt=tagBuffer.indexOf('<');
 // Ende eines tags -> weiteren in selber Zeile suchen
                }
                else  lt=-1;
 // Schleife beenden wenn tag nicht in dieser Zeile aus
            }
            System.out.print(res);
            line=buf.get();
        }
        if(!buf.isEOF())
   System.out.print("Error during reading from buffer");
    }
}
```

Unterbrechbare Aktivitäten:

```
/*
 * @(#)CancellableThread.java
 * @author Michael Sonntag
 * @version 1.0
 */

import java.lang.Thread;



public abstract class CancellableThread extends Thread
{
    private boolean cancelled=false;

    // Divers constructors ommitted

    public void cancel()
    {
        synchronized(this)
        {
            cancelled=true;
        }
        interrupt();
    }
```

```
    public boolean isCancelled()
    {
        if(cancelled)
            return true;
        synchronized(this)
        {
            return cancelled;
        }
    }

    public static boolean cancelled()
    {
        return ((CancellableThread)
(Thread.currentThread())).isCancelled();
    }

    // Why NOT a public boolean terminate(
    //                          long maxWaitToDie) ????

    public static boolean terminate(CancellableThread t,
                                    long maxWaitToDie)
    {
        // Phase 1
        if(!t.isAlive())
        {
            System.err.println("Thread already dead");
            return true;
        }
        // Phase 2
        t.cancel();
```

```
        // Phase 3
        try
        {
            t.join(maxWaitToDie*1000);
            // join is a synchronized method!
        }
        catch(InterruptedException e)
        {
        }
        if(!t.isAlive())
        {
            System.err.println("Thread cancelled");
            return true;
        }
        // Phase 4, 5 ommitted
        //Phase 6
        // Used for JDK 1.1.x. 1.2 and up should
        // use next line instead
        t.stop();
        // Not implemented up through JDK 1.1
//        t.destroy();
        // Alternative
//        t.setPriority(Thread.MIN_PRIORITY);
//        t.setDaemon(true);
        System.err.println("Thread forcibly terminated");
        return false;
    }

    public abstract void run();
}
```