# Betriebssysteme
# Studiengang Informatik / SAT

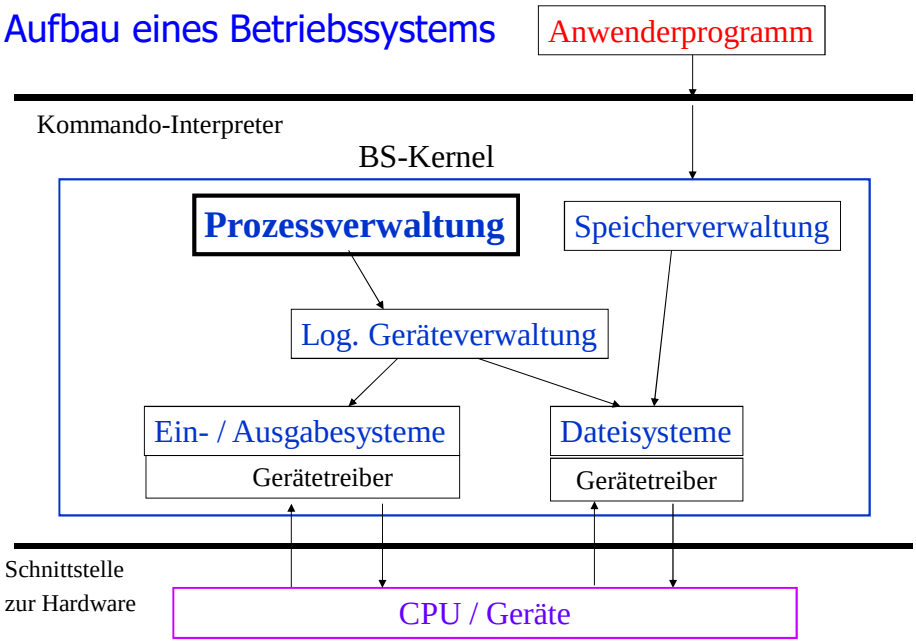
Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
Hochschule Harz
FB Automatisierung und Informatik
mwilhelm@hs-harz.de
Raum 2.202
Tel. 03943 / 659 338



## Gliederung

1. Einführung
2. Speicherverwaltung
3. Dateisysteme
4. Unix, Linux
5. **Prozesse, Thread**
6. Deadlocks

## Aufbau eines Betriebssystems






## Prozesse

- Prozessdefinition
- Prozess-Scheduling
- Multitasking
- Threads
- Synchronisationsprobleme
- Nachrichtenaustausch zwischen Prozessen

## Prozesskonzept

**Durch welche Ereignisse wird ein Prozess durch das BS gestartet?**

- Initialisierung des Systems
- Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess
  - ⌫ Hintergrundprozesse
- Benutzeranfrage, einen neuen Prozess zu erzeugen
  - ⌫ cmd, start, Explorer, Ausführen, exec, fork
- Benutzeranfrage, einen neuen Prozess zu erzeugen
- Initiierung einer Stapelverarbeitung (Batch-Job)



## Erzeugen eines Programms

**Programm erzeugen**

- Quellcode schreiben (Java, C, C++, C#, Delphi, VB, ...)
- Übersetzen
- Ergebnis Objektmodul / classmodul
- Hinzufügen von eigenen Objekten (Lib), Linker
- Hinzufügen von System-Objekten (Lib), Linker
- Ergebnis: Exe-Programm auf Platte

## Prozesskonzept

**Start eines Anwenderprogramms:**

- Programm wird mit Kommando gestartet
- Adressiert über Laufwerk/Pfad/Namen die Datei
- Schafft Speicher für das Programm/Daten/Heap
- Lädt die Datei (Hauptspeicherbereiche)
- Aktualisiert die Prozess-Verwaltung
- Analysiert den PCB
- Setzt den Instruction-Pointer IP
- Startet das Programm



## Prozesskonzept

**Ende eines Anwenderprogramms:**

- Normales Beenden
- Beenden auf Grund eines Fehlers (freiwillig)
- Beenden auf Grund eines schwerwiegenden Fehlers (unfreiwillig)
- Beenden durch einen anderen Prozess (unfreiwillig)

BS-Kommandos: exit, exitProzess

## Prozesskonzept

**Prozess**:
Ein ausgeführter Programmabschnitt mit Programmdaten, Stack, Programmzähler, Speichertabellen, Stackpointer, Registern und Flags

Prozesse werden unterbrochen und ein anderer rechenbereiter Prozess wird aktiviert

Die Prozesstabelle beschreibt alle Prozesse

Prozessbaum: logischer Zusammenhang der Kindprozesse



## Prozesstabelle in UNIX

1) Prozessverwaltung
- Register
- Programmzähler
- Programmstatuswort
- Stack-Zeiger
- Prozesszustand
- Prozesserzeugungszeitpunkt
- verbrauchte Prozessorzeit
- Prozessorzeit der Kinder
- Zeitpunkt des Alarms
- Zeiger auf Nachrichten
- unbearbeitete Signale
- Prozessnummer

2) Speicherverwaltung
- Zeiger auf Codesegment
- Zeiger auf Datensegment
- Zeiger auf BSS-Segment, ...
- Bitmaske für Signale

3) Dateiverwaltung
- UMASK
- Wurzelverzeichnis
- Arbeitsverzeichnis
- Deskriptoren, UID, GUI
- Systemaufrufparameter

## Prozess: Systemaufrufe

Ein Systemaufruf führt zu einem trap (Softwareunterbrechung), der mit einer höheren Berechtigung, Kernelzugriff, ausgeführt wird.

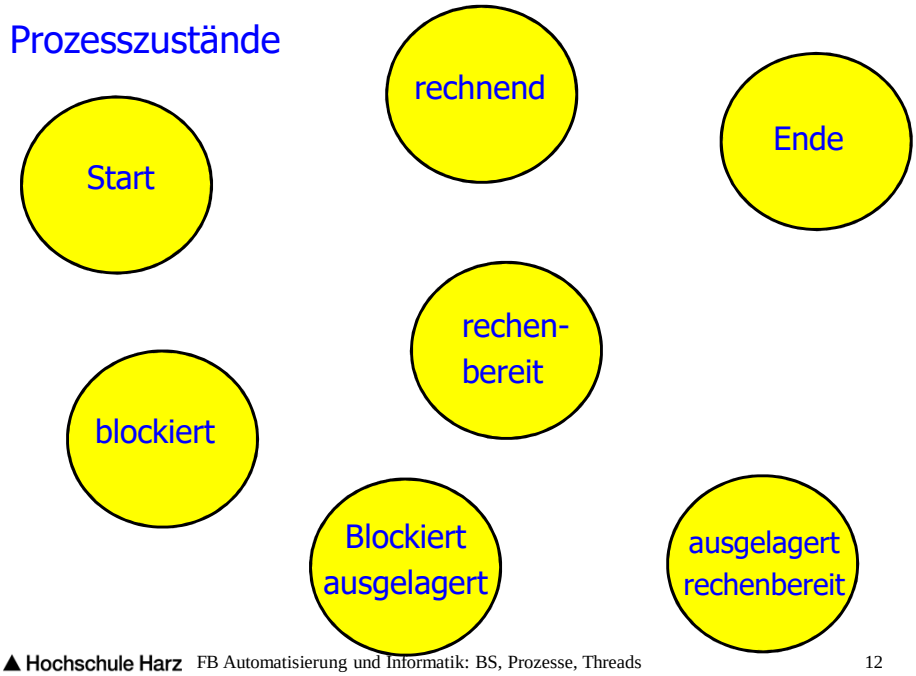
Auch der Aufruf einer Funktion des „application program interface“ (API) führt zu einem Trap

Beispiel:
Lesen von Daten aus einer Datei
der Aufrufer stellt Speicher zur Verfügung

```
count = read(file, buffer, nbytes);
```



## Prozesszustände



## Zustandsübergänge

1. Prozess wartet auf Eingabe, wird blockiert
2. Scheduler wählt einen anderen Prozess aus
3. Scheduler wählt diesen Prozess aus
4. Eingabe ist verfügbar

Betriebssystemmodul **Scheduler** wird unterhalb der Prozesse eingerichtet

Entkopplung z.B. von:

- Benutzerprozessen
- Plattenbehandlung
- Terminalprozess



## Arten des Multitasking

- **Kooperatives oder non preemptives Multitasking**
  - Windows 3.1, Macintosh OS (bis 6)
  - freiwilliges Aufrufen des Dispatchers bei jedem Systemaufruf (API)
  - freiwillig Pause durch den Aufruf des Dispatchers dann kommt der Scheduler
  - fehlerhafte Programme können das ganze System blockieren
  - keine Echtzeitfähigkeit
- **Preemptives Multitasking**
  - Unterbrechung extern veranlasst (BS)
  - Betriebssystem benutzt geeignete Auswahlkriterien
  - Zeitgeber typisch 1 bis 20ms
  - Einsatz:
    - LINUX
    - Windows NT
    - Virtual DOS Machine (VDM)

## Prozessscheduling

Gibt es mehr Bedarf an Betriebsmittel, als vorhanden ist, so muss der Zugriff koordiniert werden

### Fall 1: Prozessorverteilung auf einer Einprozessormaschine

Zielkriterien:

- Auslastung der CPU (CPU effizient benutzen)
- Durchsatz Zahl der Jobs / Zeit
- Faire Behandlung Jeder Benutzer gleiche CPU-Zeit
- Ausführungszeit Ausführungszeit pro Job minimal
- Wartezeit Wartezeit pro Job minmal
- Antwortzeit Antwortzeit pro Job minimal

Verfahren und Beurteilung:

1. Verfahren: kurze Prozesse werden bevorzugt.
2. Verfahren: Auslastung,. Ausführungszeit erhöht
3. Verfahren: Auslastung,. Ausführungszeit verringert



## Prozessscheduling, preemptives Scheduling

•Round Robin (RR)

Einfachste Strategie: Alle gleichberechtigt, Nacheinander, Unterbrechung nach Intervall

Implementation: Liste aller Prozesse
Werden nacheinander aufgerufen
Alle Prozesse sind gleich wichtig !

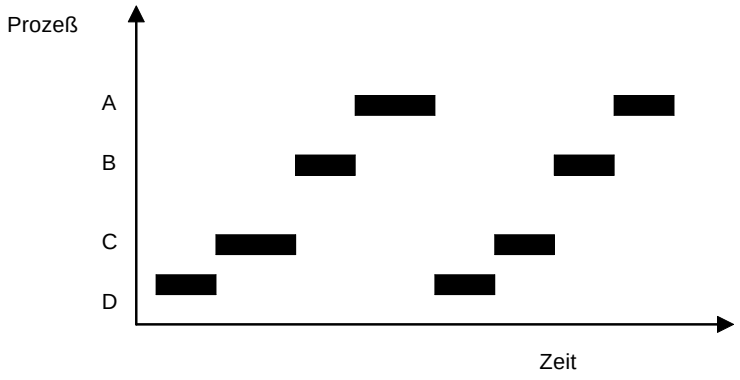
Die Antwortzeiten sind proportional zu den Bedienzeiten

Interessanter Punkt: Länge des Zeitscheibenintervalls
Umschaltung kostet Prozessorzeit

Zeitintervall zu klein → höhere Verwaltung, Eff. Durchsatz der CPU wird kleiner
Wartezeit wird größer

Zeitintervall zu groß → geringere Verwaltung, Eff. Durchsatz der CPU wird größer
Antwortzeit wird im Mehrbenutzerbetrieb wird größer

## Programmwechsel beim preemptiven Multitasking






## Prozessscheduling, Preemptives Scheduling

- Priority Scheduling (PS)

Einstufige Warteschlange.
Aus der Priorität der Prozesse wird die Position in der Warteschlange bestimmt.

Dynamischer Prozess
Verhungern ?.

**Abhilfe:**
Nach jedem Zyklus wird die Priorität des aktuellen Prozesses verringert
Prozesse mit niedrigen Prioritäten erhalten nach einer Zeitspanne $\Delta t$ höhere Prioritäten.

## Prozessscheduling, Preemptives Scheduling

### Dynamic Priority Round Robin (DPRR)

**N-stufige Warteschlange.**

- Alle Prozesse werden in Prioritätsklassen eingeteilt. Die Prozesse in der höchsten Prioritätsklasse erhalten die CPU.
- **Variante a:**
  Prozess läuft bis ein anderer Prozess eine höhere Priorität erhält
- **Variante b:**
  Prozess läuft bis ein anderer Prozess eine höhere Priorität erhält oder bis zu einem Zeitquantum
- Prioritäten können statisch oder dynamisch ermittelt werden.
- Unter Unix kann man die Priorität ändern (nice)
- **Variante c:**
  Alle Prozesse in einer Prioritätenklasse werden mit Round-Robin verwaltet. Problem des Verhungern.
- Die Priorität der Prozesse in der Vorstufe werden stetig erhöht. Ab einem Schwellenwert werden die Prozesse in die Hauptwarteschlange eingereiht.

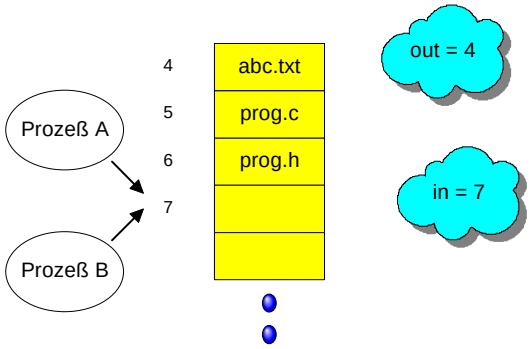


## Prozesse vs. Threads

- Ein Prozess ist ein Programm in Ausführung.
- Jeder Prozess hat eine Prozesstabelle
- Ein Prozess ist <u>kein</u> aktives Programm. Es stellt nur ein Verwaltungsobjekt dar. Ein Prozess beinhaltet aber einen Thread
- Aber: Jeder Prozess hat mindestens einen Thread (Task).
- Jede Thread kann weitere Threads erzeugen

## Zeitkritische Abläufe

Beispiel: Spooling von Dateien auf den Drucker
Spool-Verzeichnis mit beliebig vielen auszugebenden Dateien.
Die Variable *in*, gibt die Nummer des ersten freien Eintrags an
Die Variable *out*, gibt die Nummer der aktuell gedruckten Datei an.






## Spooling Problem

### Prozess A:

nr = hole freien Counter
speichert sie lokal (local_in := nr)
Druckerarray[nr] := A.LST
inkrementiert local_in
*in* := local_in

### Prozess B:

nr = hole freien Counter
speichert sie lokal (local_in := nr)
Druckerarray[nr] := B.LST
inkrementiert local_in
*in* := local_in

## Spooling Problem

Zwei Prozesse wollen eine zu druckende Datei eintragen:

Prozess A liest *in*, speichert sie lokal in local_in
Taskwechsel!
Prozess B liest *in*, speichert 7, trägt Datei B.LST ein,
inkrementiert, schreibt 8 nach *in*
Taskwechsel
Prozess A fährt fort, trägt Datei A.LST ein,
inkrementiert local_in, schreibt 8 nach *in*.

Problem: Der Platz für B.LST wird überschrieben, B.LST wird nie gedruckt!



## Zeitkritische Abläufe

Beispiel 2: Kontobewegungen, gleiches Konto

| **Zeit-punkt** | **Thread A (Einzahlung 100 €)** | **Kontostand** | **Thread B (Einzahlung 50 €)** |
|---|---|---|---|
| 1 | | 200 € | i = Kontostand |
| 2 | Prozesswechsel | | |
| 3 | i = Kontostand | | |
| 4 | Kontostand = i+100 € | 300 € | |
| 5 | Prozesswechsel | | |
| 6 | | 250 € | Kontostand = i+50 € |

- **Problem: Verlust einer Buchung**

## Lösung für Buchungsproblem

| Zeit-punkt | Thread A (Einzahlung 100 €) | Kontostand | Thread B (Einzahlung 50 €) | Sync. Objekt |
|---|---|---|---|---|
| 1 | Warte auf Sync Ob-jekt | 200 € | Warte auf Sync Objekt | |
| 3 | | | Sync start | mit thread B |
| 4 | | | i = Kontostand | mit thread B |
| 5 | | | Kontostand = i + 50 € | mit thread B |
| 6 | | 250 € | sync stop | verfügbar |
| 7 | Sync start | | | |
| 8 | i = Kontostand | | | mit thread A |
| 9 | Kontostand =i+100 € | 350 € | | mit thread A |
| 10 | Sync. stop | | | verfügbar |



## Kritische Bereiche

- **Wechselseitiger Ausschluss** muss zu manchen Zeitpunkten garantiert werden.
- Falls es erreicht werden kann, dass zu keiner Zeit zwei Prozesse in einem **kritischen Bereich** sind, können sie nebeneinander laufen.
- Betriebsmittelverwaltung ist Aufgabe des Betriebssystems!
- Eine gute Lösung, auch für parallele Programme ist:
  - Nur ein Prozess darf sich zu jedem Zeitpunkt in seinem kritischen Bereich befinden.
  - Es dürfen keine Annahmen über die Ausführungsgeschwindigkeit oder die Anzahl der Prozessoren gemacht werden.
  - Kein Prozess, der sich nicht in seinem kritischen Bereich befindet, darf andere Prozesse blockieren.
  - Kein Prozess soll unendlich lange warten müssen, bis er in seinen kritischen Bereich eintreten kann.

## Realisierung der Synchronisation

- Semaphore (Up / Down)
- Ereigniszähler (Advance, Await)
- Nachrichtenaustausch (Send, Receive)
- Monitore (Datenschutz)
  - ⌫ in Java verfügbar durch „synchronized“



## Kritische Bereich in Java

- Eine Methode kann als synchronized gekennzeichnet werden, wodurch das zugehörige Objekt gegen weitere Zugriffe anderer Threads gesperrt ist, falls diese dieselbe Methode oder eine andere Methode des Objekts aufrufen (welche ebenfalls als synchronized gekennzeichnet ist).

```
class Konto {
    // der Kontostand ist zugriffsgeschützt
    private double _Kontostand;

    //Konstruktor legt den Saldo des Kontos fest (nicht synchronized)
    public Konto(double initSaldo) {
     _Kontostand =initSaldo;
    }
```

## Synchronisation in Java

```
//Bestimmung des Saldo (muß synchronized sein)
public synchronized double getSaldo() {
   return _Kontostand;
}
//Einzahlung vornehmen (muss synchronized sein)
public synchronized double Einzahlung(double Betrag) {
 _Kontostand += Betrag;
 return _Kontostand;
}
//Auszahlung vornehmen (muss synchronized sein)
public synchronized double Auszahlung (double Betrag) {
 _Kontostand -= Betrag;
  return _Kontostand;
}  }
```



## Threads in Windows NT/XP/7

- Threads werden in Windows NT durch eine eigene Funktion realisiert, die parallel zur normalen Programmausführung arbeitet.
- Threads werden in Java durch eigene Klassen unterstützt. Der Aufwand zur Erzeugung ist dadurch minimal.
- Die Priorität der Threads ist vom Prozess übernommen bzw. highest, above normal, normal, below normal, idle
- Synchronisationsoperationen bzw. -objekte in Windows NT
  - Events
  - Mutex
  - Semaphore
  - Critical Sections

## Threads in Java

**Klassen abgeleiteten von der Klasse Thread:**

- Ableiten von der Klasse Thread
- Implementiere die run() Methode (einzige Methode des Interface)
- Deklariere ein Thread Objekt als Attribut der Klasse
- Erzeuge ein Thread Objekt und rufe die Methode start() auf
- Beende das Thread Objekt durch Aufruf der Methode stop()



## Thread-Beispiel: Deklaration

```
class TestThread extends Thread {

  public TestThread () {
    setPriority(Thread.MIN_PRIORITY);
  }

  public void run() {
    // Aktion
  }
}
```

## 1. Thread-Beispiel: Bestimmen der Summe

**Aufgaben:**

- Starten von drei Threads
- Jeder berechnet die Summe von 1 bis 100
- Methode getSumme() liefert die Summe
- Im Hauptdialogfenster wird die Summe angezeigt
- Summe von 1 bis 100 = 5050
- 5050 * 3 = 15150

$$s = \sum_{i=1}^{100} i$$



## 1. Thread-Beispiel: Bestimmen der Summe

```
public class Thread01 extends JFrame {

  ThreadSumme f1,f2, f3;
  show();
  f1 = new ThreadSumme(n);
  f2 = new ThreadSumme(n);
  f3 = new ThreadSumme(n);
  f1.start();
  f2.start();
  f3.start();
  int Summe = 0;
  Summe = f1.getSumme()+f2.getSumme()+f3.getSumme();
  label1.setText( Integer.toString(Summe) );
 } // create
```

## 1. Thread-Beispiel: Bestimmen der Summe

```
class ThreadSumme extends Thread {
  private int _n;
  private int _Summe;

  public ThreadSumme (int n) {
     _n = n;
  }
  public void run() {
    _Summe = 0;
   for (int i=1; i<=_n; i++) {
     _Summe+=i;
     delay(5);
   }
 }
```



## 1. Thread-Beispiel: Bestimmen der Summe

**Ergebnis:**

- Die drei Threads werden gestartet
- Jeder berechnet die Summe von 1 bis 100 = 5050
- Die Summe mit der Methode getSumme() liefert nicht die korrekte Summe
- Problem:
- Die einzelnen Thread sind noch nicht fertig
- Korrekte Lösung in Thread02

## 2. Thread-Beispiel: Bestimmen der Summe

```
    f1 = new ThreadSumme(n);
    f2 = new ThreadSumme(n);
    f3 = new ThreadSumme(n);
    f1.start();
    f2.start();
    f3.start();
    try {
      f1.join();
      f2.join();
      f3.join();
    }
    catch (InterruptedException e) {
    }
    int Summe = f1.getSumme()+f2.getSumme()+f3.getSumme();
    label1.setText( "Summe: "+Integer.toString(Summe) );
  } // create
```



## 2. Thread-Beispiel:

```
class ThreadSumme extends Thread {
  int nummer=0;
  JLabel myLabel;

  // Übergabe "globales JLabel"
  public ThreadSumme (JLabel label) {
     myLabel = label;
     myLabel.setText( "hallo" );
  } // create

   // Thread Methode
  public void run() {
    for (int i=1; i<=100; i++) {
      myLabel.setText( Integer.toString(i) );
      delay(50);
    }  // for
  } // run
}
```

## 3. Thread-Beispiel: Hochzählen in Threads

**Aufgaben:**

- Starten von zwei Threads
- Jeder zählt von 1 bis n
- Als Parameter im Konstruktor wird ein Verweis auf ein JLabel übergeben
- In der Run-Methode wird dieses Label aktualisiert



## Thread03

## 3. Thread-Beispiel:

```
class ThreadSumme extends Thread {
  JLabel _myLabel;
 // Übergabe "globales JLabel"
 public ThreadSumme (JLabel label) {
   _myLabel = label;
   _myLabel.setText( "hallo" );
 }
  public void run() {
   for (int i=1; i<=100; i++) {
    _myLabel.setText( Integer.toString(i) );
    try {
      Thread.sleep(50);
    }
    catch (InterruptedException e) {
    }
   }
 }
}
```



## 4. Thread-Beispiel: Bestimmen der Summe

**Aufgaben:**

- Starten von vier Threads
- Jeder Thread läuft von 1 bis 10
- Bei jedem Durchlauf wird die zugeordnete Nummer zu einem globales Label addiert
- Dazu wird dieser Inhalt gelesen und in eine Zahl umgewandelt
- Dann zu dieser Summe die Nummer addiert und zurückgeschrieben
- Es sollte also die Summe zehnmal {1,2,3,4} geben = 10*10=100
- Reihenfolge beliebig

## 4. Thread-Beispiel:

```
ThreadSumme f1,f2, f3,f4;
  f1 = new ThreadSumme(1, label);
  f2 = new ThreadSumme(2, label);
  f3 = new ThreadSumme(3, label);
  f4 = new ThreadSumme(4, label);
  summe = 0;
  f1.run();
  f2.run();
  f3.run();
  f4.run();
```



```
class ThreadSumme extends Thread {
 long _nr=0;
 JLabel _mylabel;

 public ThreadSumme (long nr, JLabel label) {
   _mylabel = label;
   _nr = nr;
 }
 public void run() {
  String s;
  long summe;
  for (int i=1; i<=10; i++) {
    s = _mylabel.getText();
    summe = Long.valueOf(s).longValue();
    summe = summe + _nr;
    _mylabel.setText( Long.toString(summe)  );
  }
 }

}
```

## 4. Thread-Beispiel:

**Ergebnis:**

- Die vier Threads werden nicht gestartet
- Jede run-Methode wird sequentiell aufgerufen
- Damit ist die Summe immer korrekt
- Statt f1.run() muss f1.start() aufgerufen werden
- Siehe TestListe5.java



## 5. Thread-Beispiel:

```
public void run() {
  String s;
  long summe;
  int time;
  for (int i=0; i<10; i++) {
    s = mylabel.getText();
    summe = Long.valueOf(s).longValue();
      // Text, Summe geholt, dann gewartet
    time = (int) (Math.random()*30);
            Thread.sleep(time);
    summe = summe + nummer;
    mylabel.setText( Long.toString(summe)  );
    try {
      Thread.sleep(10);
    }
    catch (InterruptedException e) {
    }
  }
}
```

## 5. Thread-Beispiel:

**Ergebnis:**

- Die vier Threads werden gestartet
- Jede run-Methode wird parallel aufgerufen
- Die Summe ist fast nie korrekt, da nach dem Holen immer eine Wartezeit stattfindet.
- Problem: fehlende Synchronisation



## Eigenschaften und Probleme von Threads

- Sicherheit (safety)
  - Threads nicht völlig unabhängig
  - Synchronisation
  - struktureller Ausschluss
- Lebendigkeit (lifeliness)
  - Mögliche Verklemmung, wenn man Semaphore benutzt
  - einzelne Aktivitäten sind nicht lebendig oder haben bereits aufgehört
- Nichtdeterminismus
  - die wiederholte Ausführung eines Programms braucht nicht den identischen Verlauf zu haben
  - Mangel and Vorhersagbarkeit, Transparenz
  - erschwert Fehlerbehebung
- Threads und Methodenaufruf?
  - nicht für request/reply Aufrufe geeignet (jeweils innerhalb eines Threads)
  - Callback-Funktionen erfordern, das der Thread immer aktiv ist
- Höherer Aufwand für
  - Thread-Erzeugung
  - Kontextwechsel
  - Synchronisationsaufwand

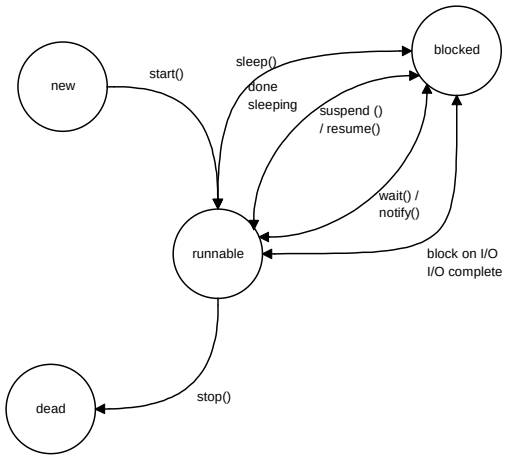
## Eigenschaften von Threads

- Steuerung
  - *start* lässt einen Thread sein run-Methode als unabhängige Aktivität aufrufen
  - *isAlive* gibt den Wert true zurück falls eine Thread gestartet aber noch nicht beendet wurde
  - *stop* beendet einen Thread unwiderruflich
  - *suspend* hält einen Thread vorübergehend an, so dass er normal weiterläuft, wenn ein anderer Thread „resume“ dieses Threads aufruft
  - *sleep* hält einen Thread für die angegebene Zeit (in Millisekunden) an
  - *join* hält den Aufrufer bis zur Beendigung des Zielthreads an.
  - *interrupt* bricht eine sleep-, wait- oder join-Methode durch eine InterruptedException ab
- Prioritäten
  - Per Voreinstellung erhält jeder Thread dieselbe Priorität
  - Mittels *Thread.setPriority* kann die Priorität zwischen Thread.MIN_PRIORITY und Thread.MAX_PRIORITY eingestellt werden
  - Ein Prozess mit höherer Priorität unterbricht evt. einen Prozess mit niedrigerer Priorität
- Warten und Benachrichtigen
  - *wait* bewirkt das Anhalten des aktuellen Threads, eintragen in die interne Warteschleife und die Synchronisationssperre wird aufgehoben
  - *notify* erreicht, dass - falls vorhanden - ein willkürlich ausgewählter Thread T aus der internen Warteschleife entfernt wird
  - *notifyAll* informiert alle Threads in der internen Warteschlange



## Zustände der Threads

- new
- runnable
- blocked
- dead
  - run wurde beendet
  - stop wurde aufgerufen
- Sonderfall: runnable/Demon



## Threads in Java

**Thread mit einer abgeleiteten Klasse:**

- Deklariere in der Klasse: implements Runnable
- Implementiere die run() Methode (einzige Methode des Interface)
- Deklariere ein Thread-Objekt als Attribut der Klasse
- Erzeuge ein Thread-Objekt und rufe die Methode start() auf
- Beende das Thread-Objekt durch Aufruf der Methode stop()



## 10. Thread-Beispiel: Zwei getrennte Summationen

Hauptfenster

1. Summe

2. Summe

## 10. Thread-Beispiel:

Main-Dialogfenster

```
  // Thread Variablen definieren
  EinzelListen f1,f2;
  // Threads erzeugen
  f1 = new EinzelListen(1);
  f2 = new EinzelListen(2);
  // Frames anzeigen
  f1.setVisible(true);
  f2.setVisible(true);
  // Threads starten
  f1.run();
  f2.run();
 } // create
```



## 10. Thread-Beispiel:

```
class EinzelListen extends JFrame implements Runnable {

 // Thread-Methode
 public void run() {
  for (int i=0; i<100; i++) {
   _label.setText( Integer.toString(i) );
    try {
      Thread.sleep(50);
    } catch (InterruptedException e) {
    }
  }
 }


} // EinzelListen
```

## 10. Thread-Beispiel:

**Ergebnis:**

- Die zwei Threads werden nicht gestartet (run)
- Jede run-Methode werden sequentiell aufgerufen
- Abhilfe: Methode start(), TestListe11.java



## 11. Thread-Beispiel: Externe Thread-Objekte

```
      // Thread Variablen definieren
EinzelListen f1,f2;
  // Threads erzeugen
f1 = new EinzelListen(1);
f2 = new EinzelListen(2);
   // Frames anzeigen
f1.show();
f2.show();
// Threads starten
Thread t1, t2;
t1 = new Thread(f1);    // Thread wird erzeugt
t2 = new Thread(f2);    // Thread wird erzeugt
t1.start();             // Thread wird gestartet
t2.start();             // Thread wird gestartet
```

## 12. Thread-Beispiel: Interne Thread-Objekte

```
public EinzelListen (int nr) {
  Thread t;

  t = new Thread(this);  // Thread wird erzeugt
  t.start();             // Thread wird gestartet
 } // create


 // Thread-Methode
 public void run() {
     // Aktion
}
```



## Realisierung der Synchronisation

- Semaphore (Up / Down)
- Ereigniszähler (Advance, Await)
- Nachrichtenaustausch (Send, Receive)
- Monitore (Datenschutz)
  - ⌫ in Java verfügbar durch „synchronized“

## Semaphore

Entwickelt 1965 von E. W. Dijkstra.
Der Schwerpunkt liegt hier im Schlafen und Wecken von Prozessen,
um so unnötige Prozessorvergeudung zu verhindern.

**Prinzip:**

- Einführung einer Integervariablen - Semaphor.
- Operation DOWN
  - Fall Semaphor > 0,
    - Semaphor wird um eins erniedrigt
    - Prozess startet
  - Fall Semaphor <= 0, Prozess wird schlafen gelegt
- Operation UP (Semaphor wird um eins erhöht),
  - Falls Semaphor=1, dann wird ein Prozess aufgeweckt (Sind Prozesse vorhanden?)
  - Falls Semaphor>1, dann passiert nichts



## Semaphore

**Prinzipieller Ablauf:**

- DOWN(P1)
- kritischer Bereich
- UP(P1)

| sem1 = 0 | |
|---|---|
| P1 | P2 |
| While (true) do<br>DOWN(sem1);<br><br>criticalSection()<br>UP(sem1);<br>noncriticalSection();<br>end | While (true) do<br>DOWN(sem1);<br><br>criticalSection()<br>UP(sem1);<br>noncriticalSection();<br>end |

## Semaphore

**Ablauf:**

- Das Semaphor p muss mit 1 initialisiert werden

| sem1 = 1 | |
|---|---|
| P1 | P2 |
| While (true) do<br>DOWN(sem1);<br><br>CriticalSection()<br>UP(sem1);<br>noncriticalSection();<br>end | While (true) do<br>DOWN(sem1);<br><br>CriticalSection()<br>UP(sem1);<br>noncriticalSection();<br>end |



## Erzeuger - Verbraucher - Problem

**Zwei Prozesse tauschen über einen gemeinsamen Puffer Daten aus:**

- Prozess E (Erzeuger) erzeugt Daten und speichert sie im Puffer ab
- Prozess V (Verbraucher) verbraucht die Daten, in dem er sie aus dem Puffer abholt.
- Geschwindigkeiten der Prozesse
  - Verbraucher ist zu schnell
  - Erzeuger ist zu schnell
- Verwendung von zwei Semaphoren
  - Erzeuger-Semaphor
  - Verbraucher-Semaphor

## Erzeuger - Verbraucher - Problem

Zwei Prozesse tauschen über <u>einen</u> gemeinsamen Puffer Daten aus:

## Erzeuger - Verbraucher - Problem

| Semaphor leer = 0<br>Semaphor voll = 0; | |
|---|---|
| P Erzeuger | P Verbraucher |
| While (true) do<br><br><Erzeuge Daten><br><br><fülle Puffer><br><br>end | While (true) do<br><br><leere Puffer><br><br><verarbeite Daten><br><br>end |

## Bounded-Buffer - Problem

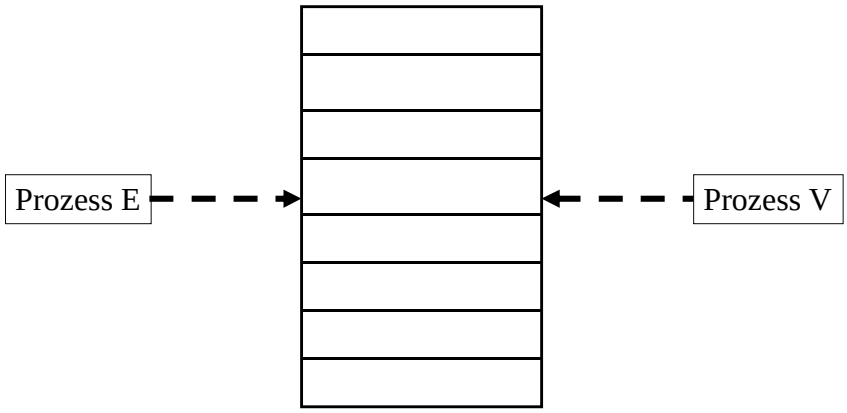
**Zwei Prozesse tauschen über einen gemeinsamen Puffer (n-Datensätze) Daten aus:**

- Prozess E (Erzeuger) erzeugt Daten und speichert sie im Puffer ab
- Prozess V (Verbraucher) verbraucht die Daten, in dem er sie aus dem Puffer abholt.
- Geschwindigkeiten der Prozesse
  - Verbraucher ist zu schnell
  - Erzeuger ist zu schnell
- Verwendung von zwei Semaphoren
  - Erzeuger-Semaphor
  - Verbraucher-Semaphor



## Bounded Buffers - Problem

Zwei Prozesse tauschen über einen gemeinsamen Puffer Daten aus:



## Bounded-Buffer - Problem

| Semaphor frei = ?;<br>Semaphor belegt = ?; | |
|---|---|
| P Erzeuger | P Verbraucher |
| While (true) do<br><br><Erzeuge Daten><br><br><in den Puffer schreiben><br><br>End | While (true) do<br><br><aus dem Puffer lesen><br><br><verarbeite Daten><br><br>end |



## Semaphore in Java: Beispielimplementierung

```
class Counter extends Threads {
    private int count;
    public int getQueueLength;
    public Counter(int initialCount) {
      count = initialCount;
    }
    public void synchronized  acquire() throws InterruptedException {
        getQueueLength++;    // Interne Warteschlange
       if (count = = 0 )        wait();
       count --;
        getQueueLength--;
    }
    public void synchronized  release() throws InterruptedException {
        count ++;
       notify();
    }
```

## Semaphore in Java

- Semaphore sind ab dem JDK 1, 5 in Java implementiert.
- Package: import java.util.concurrent.Semaphore;
- Konstruktor: Semaphore(int permits, boolean fair);

**Wichtige Methoden:**

- Methoden Down:
- public void acquire() throws InterruptedException;
- public void acquire(int permits) throws InterruptedException;
  public boolean tryAcquire(int permits);
- public getQueueLength(); public int availablePermits();
- Methoden Up:
- public void release() throws InterruptedException;
- public void release(int permits) throws InterruptedException;



## Semaphore in Java

**Beispiel:**

```
import java.util.concurrent.Semaphore;
 Semaphore sem = new Semaphore(1,true);

void P1(Semaphore sem){
  While (true) {
      Erzeuge Daten();
      sem.acquire();  // Down();
      criticalSection()
      sem.release();  // Up();
      noncriticalSection();
  }
}
```

```
void P2(Semaphore sem){
  While (true) {
      sem.acquire();  // Down();
      criticalSection()
      Hole_daten();
      sem.release();  // Up();
      noncriticalSection();
  }
}
```

## Swing und Threads

- **Die GUI-Elemente des Pakets AWT sind Threadssicher**
- **Die GUI-Elemente des Pakets Swing sind <u>nicht</u> Threadssicher**



## Warum Swing nicht Threadsicher ist

- Die Tatsache, dass das Swing-Toolkit nicht Thread-sicher ist, erstaunt vielleicht auf den ersten Blick. Das AWT ist Thread-sicher, da AWT auf Plattform-Peer-Elemente vertraut. In einer List-Box unter dem AWT ist es problemlos möglich, ein Element einzufügen und parallel zu löschen. Doch auf die Synchronisation bei Swing wurde aus zwei Gründen verzichtet:
- Untersuchungen mit anderen grafischen Bibliotheken haben ergeben, dass Operationen in Threads zu ärgerlichen Deadlock-Situationen führen können. Es ist eine zusätzliche Last, die auf dem Programmieren grafischer Oberflächen lastet, Monitore (Semaphare) korrekt einzusetzen.
- Der zweite Punkt ist der Gewinn von Ausführungsgeschwindigkeit zu nennen. Das Swing-Toolkit kostet ohnehin viel Zeit, so dass auf die zusätzliche Synchronisation gut verzichtet werden kann.

## Lösung für Swing

Einige der Methoden, die dennoch synchronisiert sind, tragen Listener ein, so etwa bei JComponent addPropertyChangeListener(), removePropertyChangeListener() und addVetoableChangeListener(), removeVetoableChangeListener(). Bei JCheckBoxMenuItem ist es dann die einsame Methode setState(boolean), die synchronisiert ist. Es findet sich intern mal hier mal da ein synchronisierter Block. Ansonsten ist jedoch nicht viel dabei, und wir müssen unsere Teile synchronisiert ausführen.

Um Programmstücke konform ausführen zu lassen, definiert Swing einige Methoden und Klassen. **Dazu gehören:**

- invokeLater(Runnable)
- invokeAndWait(Runnable)
- JProgressBar
- ProgressMonitor
- ProgressMonitorInputStream
- SwingWorker



## Lösung für Swing

Da Swing nicht Thread-sicher ist, ist die einzige Möglichkeit zur Manipulation von Oberflächenelementen der AWT-Thread.

Das Event muss in die AWT-Event-Queue eingetragen werden. Genau für diese Aufgabe existieren in der Klasse EventQueue zwei Methoden: invokeLater() und invokeAndWait(). Damit lassen sich beliebige Programmstücke in die Warteschlange einführen. In der Warteschlange für das AWT liegen Aufträge und Ereignisse, die an die Oberflächenelemente verteilt werden. Alles spielt sich dabei neben dem Haupt-Thread ab, so dass Parallelität herrscht. Hat die Warteschlange alle Ereignisbehandler aufgerufen, kann der Programmcode von invokeLater() und invokeAndWait() durchlaufen werden. Den Methoden wird ein Runnable-Objekt übergeben. Die zwei Methoden erfüllen unterschiedliche Bedürfnisse:

- invokeLater() legt einen Thread in die Warteschlage und kehrt sofort zurück. Die Methode arbeitet somit asynchron. Der Aufrufer weiß nicht, wann der Programmcode abgearbeitet wird.
- invokeAndWait() legt ebenfalls den Thread in die Warteschlange, verharrt aber so lange in der Methode, bis der Programmcode in run() aufgerufen wurde. Die Methode ist also synchron.

## Beispiel

Ein Fortschrittsbalken JProgressBar mit dem Namen bar soll in einer Schleife einer Berechnung angepasst werden.

```
For (int i=0; i<MAX;i++) {
   EventQueue.invokeLater( new Runnable() {
          public void run() {
        bar.setValue  ( i );
        }
        }
   aktion();
);
```



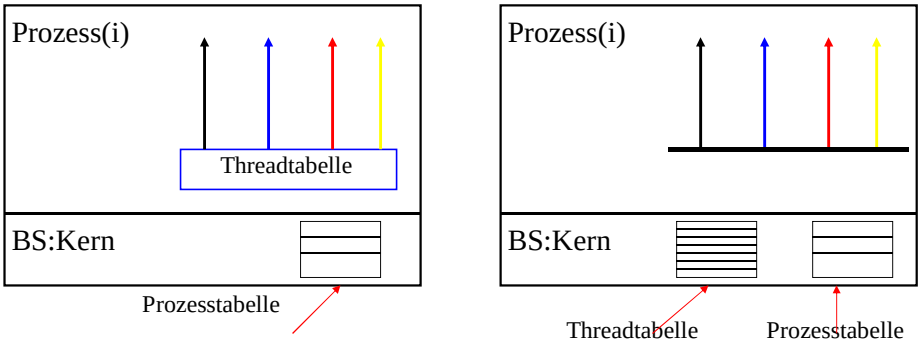
## Aus JExplorer

```
// Prozentrechnung
double d =  100.0*( (_summe2*1.0)/_summe1);
int k = (int)  (d);
if (k>0) {
   // Single Thread Rule
  final int Runnable_k=k;
  final String  Runnable_filename=myfile.Name;
  SwingUtilities.invokeLater(
   new Runnable(){
      public void run() {
         progress.setValue(Runnable_k);
         label3b.setText( Runnable_filename );
      }
   }
 );  // invokeLater
}
```

Siehe auch Pipe_02

## Implementation von Threads

- Threads können als User-Thread oder als Kernel-Thread implementiert werden.






## Abbildung der Threads

- Many to One Modell
  - n User-Thread abgebildet auf einen Kernel-Thread
  - Eigene Threadverwaltung in Userraum, Effizient, System-Call blockiert alle
- One to One Modell
  - n User-Thread abgebildet auf einen n-Kernel-Thread
  - Threadverwaltung im BS, mehr Parallelität, Aufwändiger, n CPU, Windows
- Many to Many Modell
  - n User-Thread abgebildet auf einen m-Kernel-Thread
  - zum Beispiel in Solaris, HP-UX, Tru64 Unix

## Beenden eines Threads

- **Methode stop ist deprecated**
- **Alternative: globale Variable**
- **Alternative: Methode interrupted()**



## Threads: globale Variable

```
Class myThread extends Thread {
    private static boolean _bWeiter=true;
    public run() {
        while ( _bWeiter ) {
            // Aktion
        }
        // Aufräumen
    }
}
```

Variante 1:

myThread.setWeiter(false);

Variante2:

t1.setWeiter(false);

## Threads: globale Variable

```
void starteThread() {
    myThread t1 = new myThread();
    myThread t2 = new myThread();
}
```

Variante 1:

```
    myThread.setWeiter(false);      // Alle Threads werden gestoppt
```

Variante2:

```
    t1.setWeiter(false);            // t1 wird gestoppt
```



## Threads: Methode interrupted()

```
Class myThread extends Thread {

 public void run() {
      while (  ! isInterrupted()    )
          try {
            Thread.sleep(300);
          }
          catch (InterruptedException e) {
             this.interrupt();  //
             System.out.println("Ende in der catch-Anweisung");
         }
         // eigene Aktion
     }  // while
     System.out.println("NACH DER SCHLEIFE");
 } // run

 } // myThread
```

## Threads: Methode interrupted()

```
Class myThread extends Thread {

 public void run() {
      while (  ! isInterrupted()  && _bWeiter   )
          try {
            Thread.sleep(300);
           }
           catch (InterruptedException e) {
              this.interrupt();  //
              System.out.println("Ende in der catch-Anweisung");
          }
          // eigene Aktion
     }  // while
     System.out.println("NACH DER SCHLEIFE");
} // run

} // myThread
```



## Threads: Methode interrupted()

```
void starteThread() {

    myThread [] t = new myThread[10];

    t[0] = new myThread();

    t[1] = new myThread();

}


void stopThread(int nr) {

    t[nr].interrupted();       // Nur ein Thread wird gestoppt

}
```