

# MAS: Betriebssysteme

## Prozesse und Threads

T. Pospíšek

# Gesamtüberblick

---

1. Einführung in Computersysteme
2. Entwicklung von Betriebssystemen
3. Architekturansätze
4. Interruptverarbeitung in Betriebssystemen
- 5. Prozesse und Threads**
6. CPU-Scheduling
7. Synchronisation und Kommunikation
8. Speicherverwaltung
9. Geräte- und Dateiverwaltung
10. Betriebssystemvirtualisierung

# Zielsetzung

---

- Das Prozess- und das Threadmodell verstehen und erläutern können
- Den Lebenszyklus von Prozessen und Threads innerhalb eines Betriebssystems verstehen und erläutern können

# Überblick

---

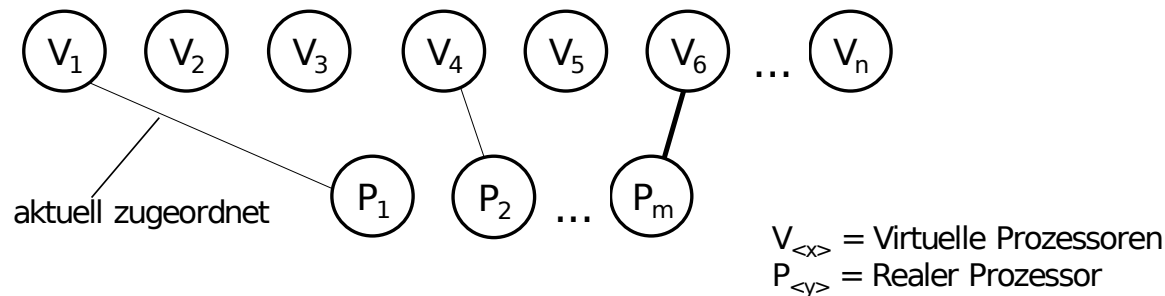
- 1. Prozesse und Lebenszyklus von Prozessen**
2. Threads
3. Threads im Laufzeitsystem

# Prozesse

- Informelle Definitionsansätze: Ein **Prozess** (manchmal auch Task genannt):
  - ist die Ausführung (Instanziierung) eines Programms auf einem Prozessor
  - ist eine dynamische Folge von Aktionen verbunden mit entsprechenden Zustandsänderungen
  - ist die gesamte Zustandsinformation der Betriebsmittel eines Programms

# Virtuelle Prozessoren

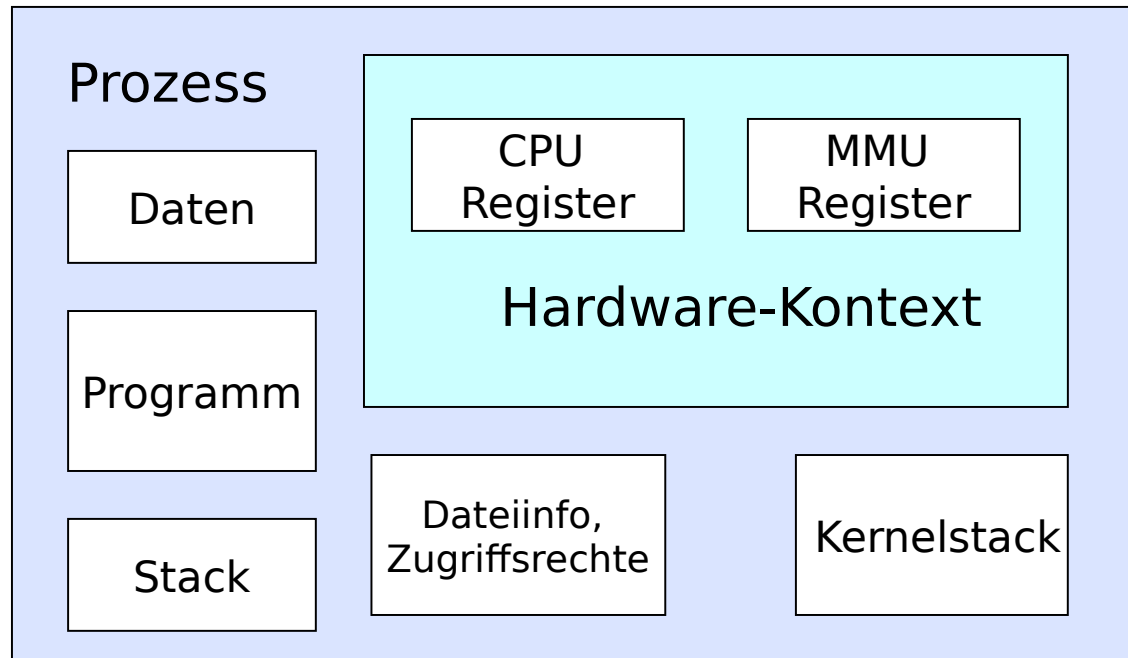
- Das Betriebssystem ordnet im Multiprogramming jedem Prozess einen **virtuellen Prozessor** zu
- Echte Parallelarbeit, falls jedem virtuellen Prozessor ein **realer Prozessor** bzw. Rechnerkern zugeordnet wird
- **Quasi parallel:** Jeder reale Prozessor ist zu einer Zeit immer nur einem virtuellen Prozessor zugeordnet und es gibt Prozess-Umschaltungen



# Prozesse und Betriebsmittel

- Prozesse **konkurrieren** um die Betriebsmittel
- Beispiel bei nur einer CPU und mehreren Prozessen:
  - Prozesse laufen abwechselnd einige Millisekunden
  - Dadurch entsteht der Eindruck paralleler Verarbeitung
  - Dazwischen sind Prozesswechsel (**Kontextwechsel** oder „context switch“)
    - Ausführung des bisherigen Prozess wird unterbrochen („Prozess wird gestoppt“)
    - Ausführung eines anderen Prozesses wird fortgeführt („neuer Prozess wird (re)aktiviert“)

# Prozesskontext



MMU = Memory Management Unit

- Prozesskontext = gesamte Zustandsinformation zu einem Prozess
- Kernelstack = Stack für Systemaufrufe des Prozesses



# Prozesskontext

```
#include <stdio.h>
```

```
int glob_count = 0;
```

```
int etwas_machen(int *count)
{
    int i;
    ...
}
```

```
int main()
{
    etwas_machen(&glob_count);
    return 0;
}
```

Register

SP  
PC  
GP0  
GP1  
...

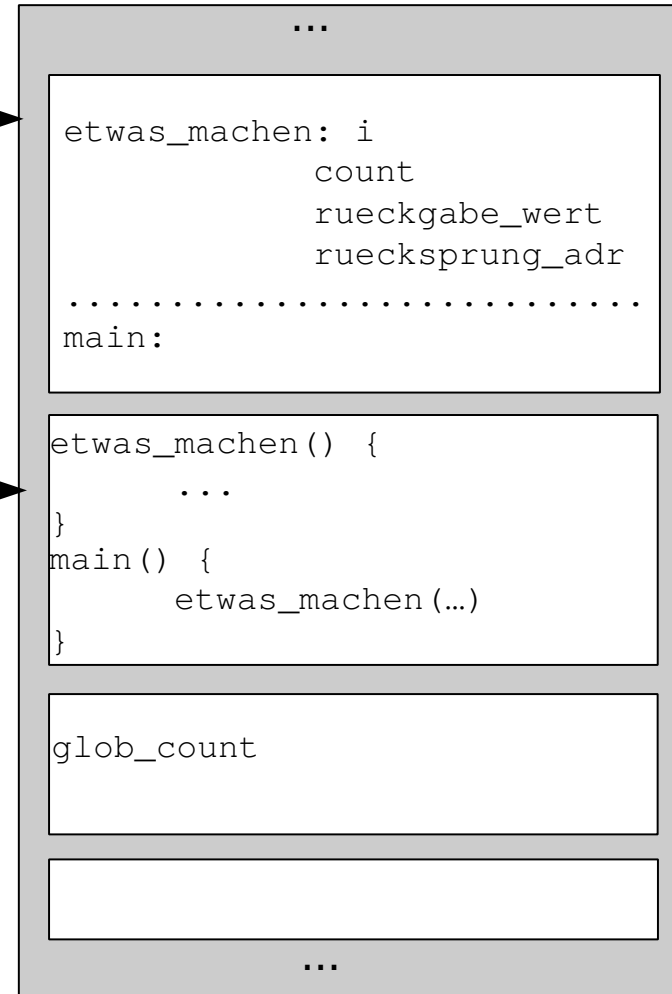
Identität

PID  
UID  
GID  
...

Ressourcen

Open  
Files  
Sockets  
Locks  
...

Virtueller Adressbereich



Tiefste  
Adresse  
Stack

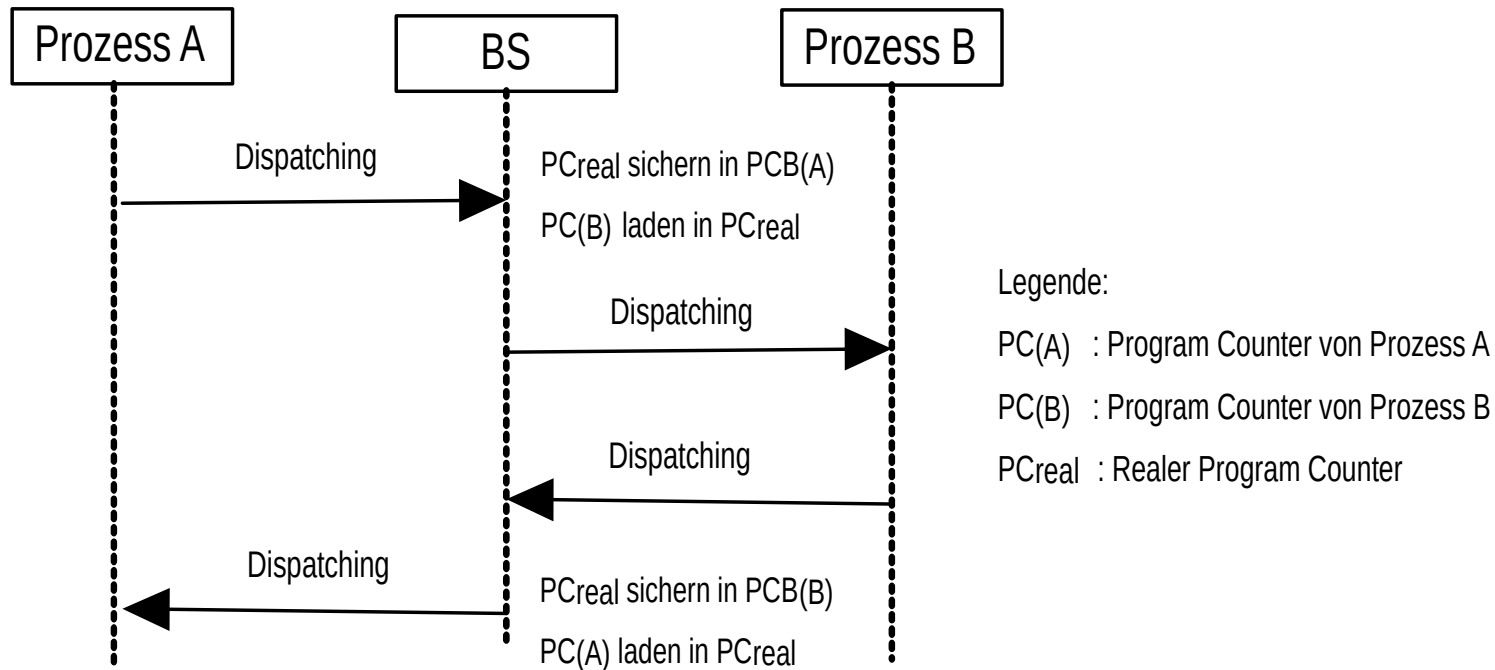
Text /  
Instrukt.

Data

Heap

Höchste  
Adresse

# Prozesskontextwechsel



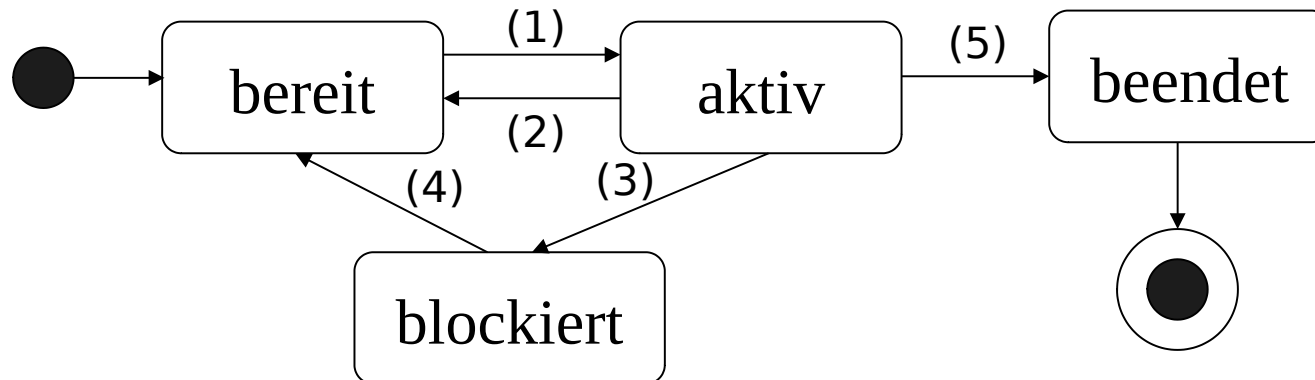
- PCB – Process Control Block
- Hardware-Kontext von Prozess A in seinen PCB sichern
- Gesicherten Hardware-Kontext von Prozess B aus seinem PCB in die Hardware (Ablaufumgebung) laden

# Prozesslebenszyklus

- Ein Prozess wird mit Mitteln des Betriebssystems erzeugt, Beispiel in Unix: Systemaufruf **fork()**
  - Realen Prozessor, Hauptspeicher und weitere Ressourcen zuordnen
  - (Programmcode und Daten in Speicher laden) („copy on write“)
  - Prozesskontext laden und Prozess starten
- Für das Beenden eines Prozesses gibt es mehrere Gründe:
  - Normaler exit
  - Error exit (vom Programmierer gewünscht, fatal error)
  - Durch einen anderen Prozess beendet (killed)

# Prozesslebenszyklus: Zustandsautomat eines Prozesses

- Prozesse durchlaufen während ihrer Lebenszeit verschiedene Zustände (Zustandsautomat):



- (1) Betriebssystem wählt den Prozess aus (Aktivieren)
- (2) Betriebssystem wählt einen anderen Prozess aus (Deaktivieren, preemption, Vorrangunterbrechung)
- (3) Prozess wird blockiert (z.B. wegen Warten auf Input, Betriebsmittel wird angefordert)
- (4) Blockierungsgrund aufgehoben (Betriebsmittel verfügbar)
- (5) Prozessbeendigung oder schwerwiegender Fehler (Terminieren des Prozesses)

# Prozesstabelle und PCB

- Betriebssystem verwaltet eine **Prozesstabelle**
  - Information, welche die Prozessverwaltung für Prozesse benötigt, wird in einer Tabelle bzw. mehreren Tabellen/Listen verwaltet
- Ein Eintrag in der Prozesstabelle wird auch als Process Control Block (**PCB**) bezeichnet
- Einige wichtige Informationen im PCB
  - Programmzähler
  - Prozesszustand
  - Priorität
  - Verbrauchte Prozessorzeit seit dem Start des Prozesses
  - Prozessnummer (PID), Elternprozess (PID)
  - Zugeordnete Betriebsmittel, z.B. Dateien (Dateideskriptoren)

# Prozessverwaltung unter Unix: Prozesshierarchie und init-Prozess

- Unix besitzt eine **baumartige** Prozessstruktur (Prozesshierarchie)
- Jeder Prozess erhält vom Betriebssystem eine **PID** (eindeutige Prozess-ID)
- Besondere Prozesse unter Unix:
  - **scheduler** (PID 0), früher: **swapper**-, auch **idle**-Prozess genannt, je nach Betriebssystem
    - Speicherverwaltungsprozess für Swapping (später mehr dazu)
  - **init** (PID 1), bei Mac OS X heißt der Prozess **launchd**
    - Urvater aller Prozesse

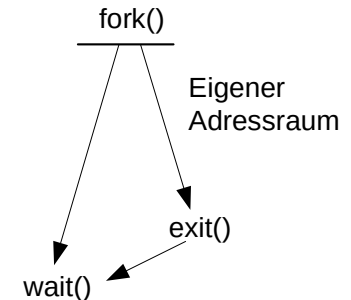
# Prozessverwaltung unter Unix: Prozesserzeugung - fork

- Ein Prozess wird unter Unix durch einen ***fork()***-Aufruf des Vaters erzeugt
- Der Kindprozess wird erzeugt und erbt dessen Umgebung **als Kopie**:
  - Alle offenen Dateien und Netzwerkverbindungen
  - Umgebungsvariablen
  - Aktuelles Arbeitsverzeichnis
  - Datenbereiche
  - Codebereiche
- Durch den System-Call ***execve()*** kann im Kindprozess ein neues Programm geladen werden

# Prozesserzeugung unter Unix (C-Beispiel)

```
static void main()
{
    int ret;                // Returncode von `fork`.
    int status;             // Status des Kindprozesses.
    pid_t pid;              // pid_t ist ein spezieller Datentyp, der eine PID
    beschreibt.
    ret = fork();         // Erzeuge Kindprozesses.
    if (ret == 0) {
        // Anweisungen, die im Kindprozess ausgeführt werden.
        ...
        exit(0);           // beende den Kindprozesses mit Status 0 (ok)
    }
    else {
        // Anweisungen, die nur im Elternprozess ausgeführt werden.
        // Zur Ablaufzeit kommt hier nur der Elternprozess rein.
        // ret = PID des Kindprozesses

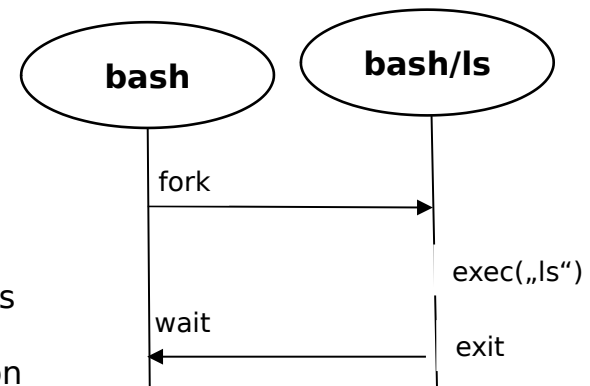
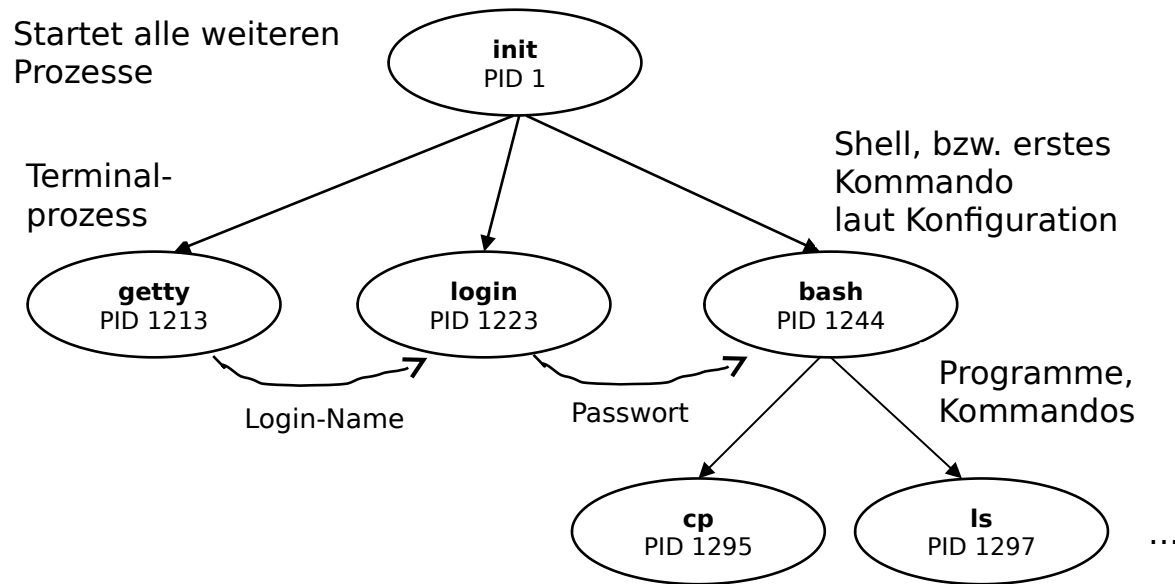
        ...
        pid = wait(&status); // warte auf Beendigung des Kindprozesses
        exit(0);             // beende Vaterprozesses mit Status 0 (ok)
    }
}
```





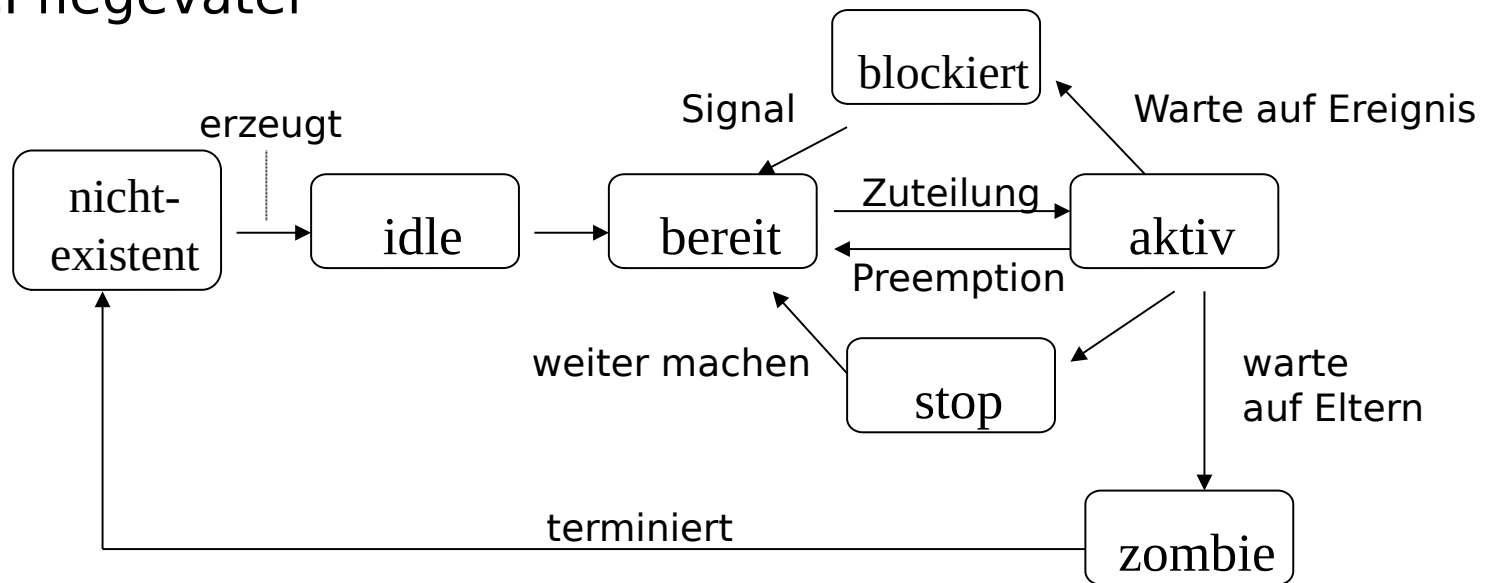
# Unix-Prozessbaum

- Je Terminal wartet ein `getty`-Prozess auf eine Eingabe (Login)
- Nach erfolgreichem Login wird ein Shell-Prozess gestartet
- Jedes Kommando wird gewöhnlich in einem eigenen Prozess ausgeführt
- `ps tree` oder `ps faux` für Prozessbaum Anzeige



# Zustandsautomat eines Unix-Prozesses

- Jeder Prozess, außer der init-Prozess, hat einen Elternprozess
- Zustand *zombie* wird vom Kindprozess eingenommen, bis der Elternprozess Nachricht über Ableben erhalten hat
- Elternprozess stirbt vorher -> init-Prozess wird „Pflegevater“

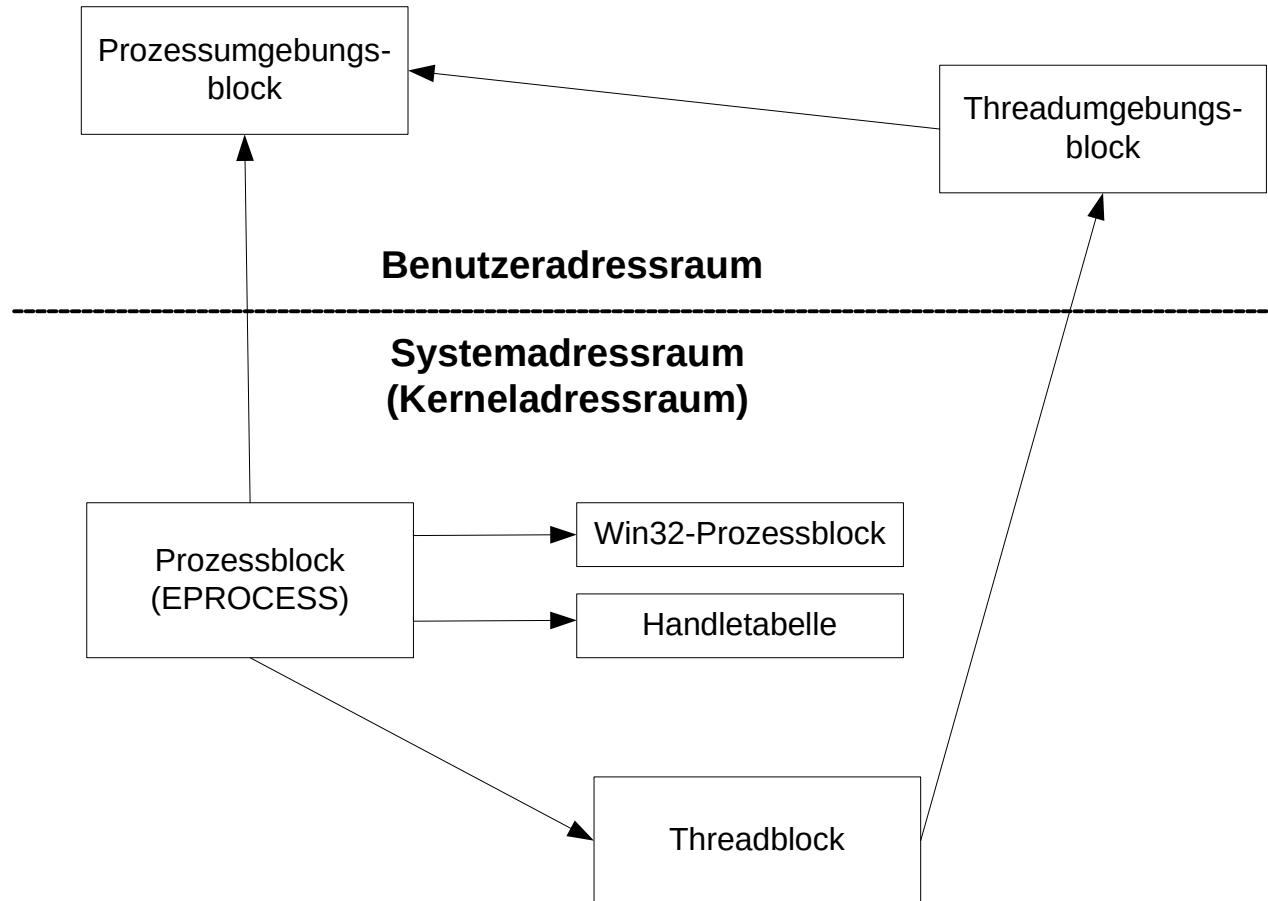


*idle* und *zombie* sind Zwischenzustände

# Prozessverwaltung unter Windows

- Die Prozesserzeugung ist in Windows komplexer als unter Unix
- System Call **CreateProcess()** dient der Erzeugung von Prozessen
- Jeder Prozess erhält zur Verwaltung ein Objekt-Handle mit **PID** (Idle-Prozess hat PID 0)
- **POSIX-fork()**-Mechanismus geht auch unter Windows (in einem POSIX-Prozess) und wird auf *CreateProcess()* abgebildet

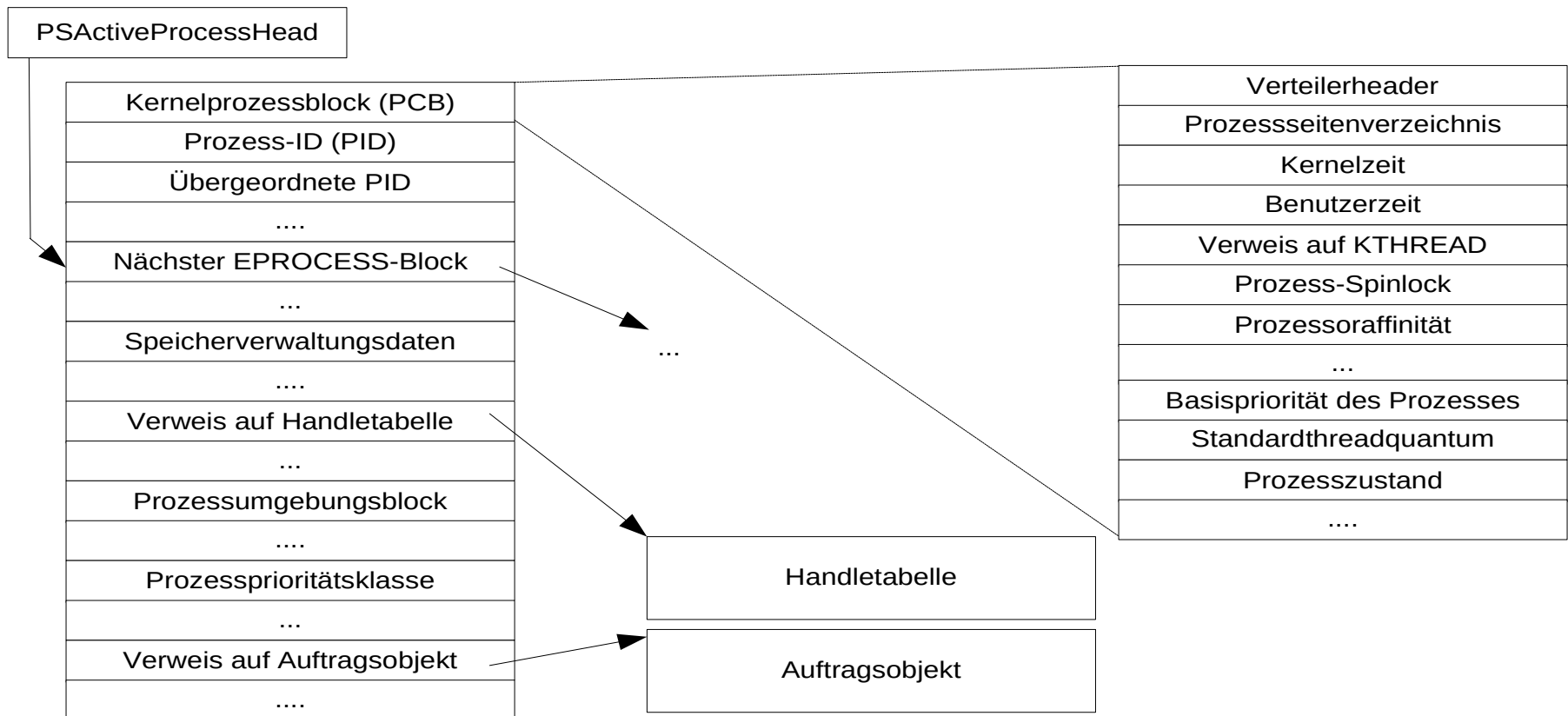
# Datenstrukturen unter Windows



Quelle: Solomon, D. A.; Russinovich, M.: Microsoft Windows Internals, Microsoft Press, Part 1 und 2, 6. Auflage, 2013

# Der EPROCESS-Block unter Windows

- Der EPROCESS-Block enthält wichtige Informationen zum Prozess



Quelle: Solomon, D. A.; Russinovich, M.: Microsoft Windows Internals, Microsoft Press, Part 1 und 2, 6. Auflage, 2013

# Überblick

---

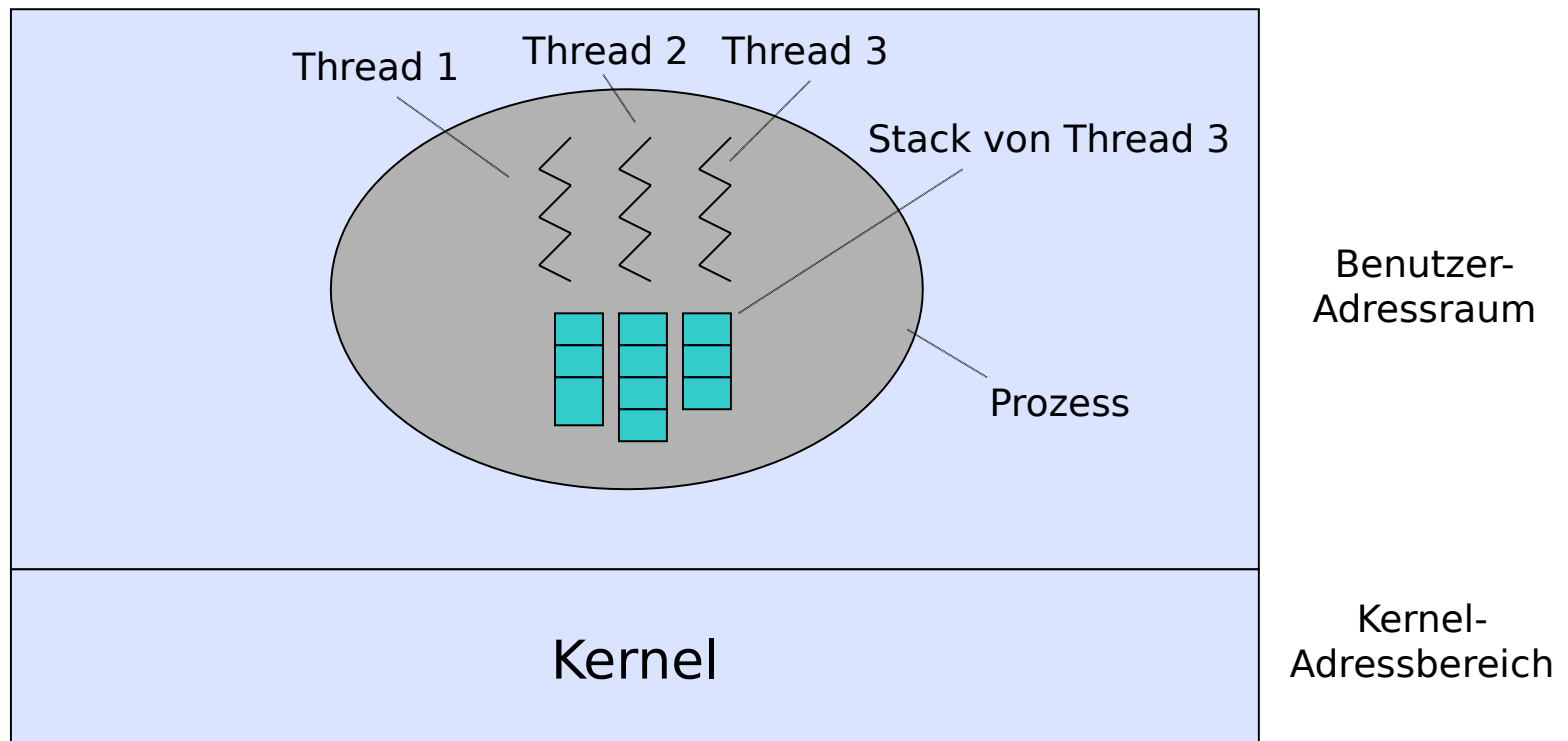
1. Prozesse und Lebenszyklus von Prozessen
- 2. Threads**
3. Threads im Laufzeitsystem

# Threads

- **Leichtgewichtige** Prozesse (lightweight processes, LWP)
- Gemeinsame Ressourcen im Prozess:
  - **Gemeinsamer Adressraum**
  - Offene Files, Netzwerkverbindungen ...
- Eigener **Zustandsautomat** ähnlich wie Prozess
- Mehrere Threads im Prozess → **Multithreading**
- Threads können auf Benutzerebene oder auf Kernelebene implementiert werden
- Threads sind nicht gegeneinander geschützt
  - Synchronisationsmaßnahmen erforderlich

# Threads, Stack

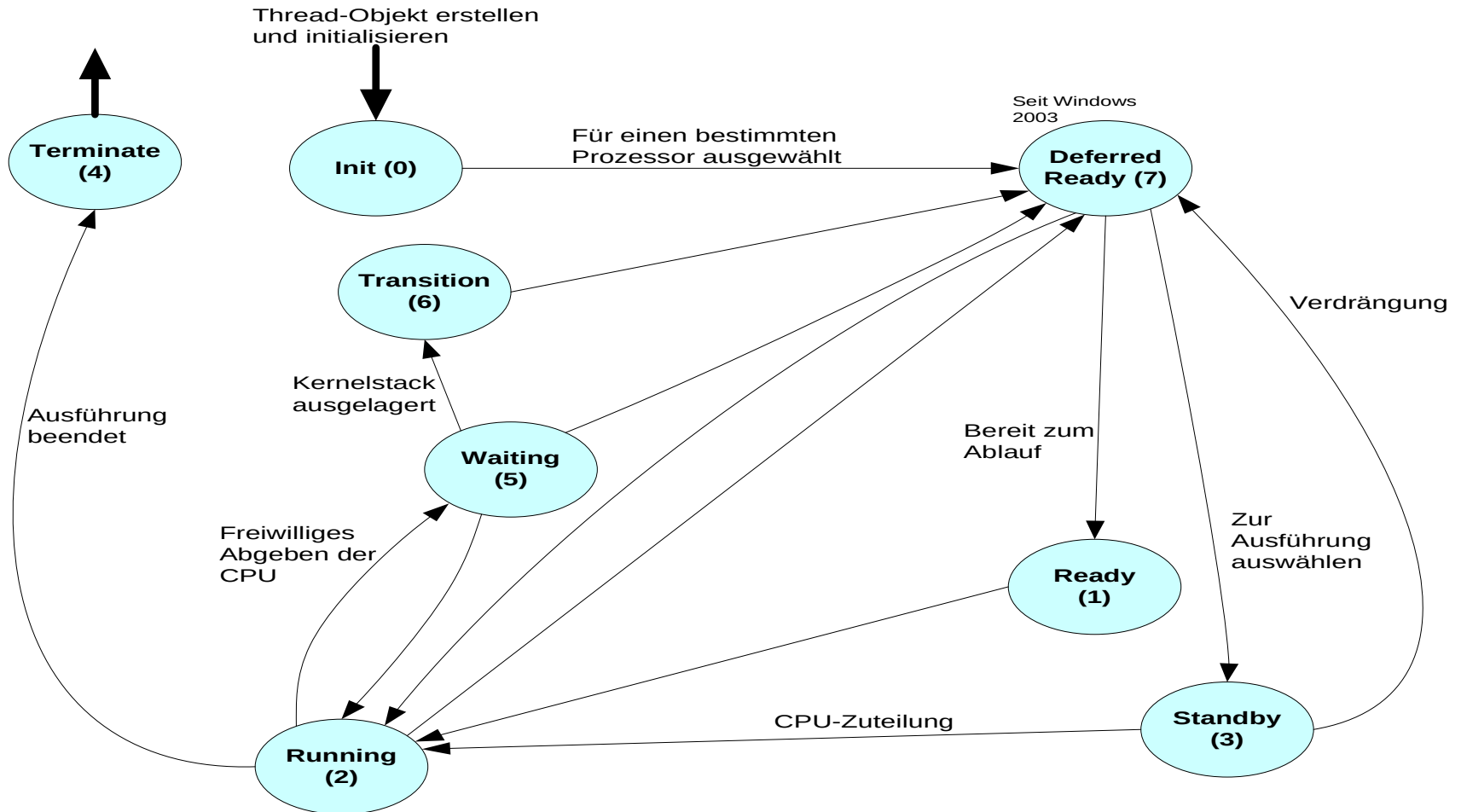
- Threads haben einen eigenen Programmzähler, einen eigenen log. Registersatz und einen eigenen Stack



Quelle: Tanenbaum, A. S.: Moderne Betriebssysteme, 3. aktualisierte Auflage, Pearson Studium, 2009



# Thread-Zustandsautomat unter Windows

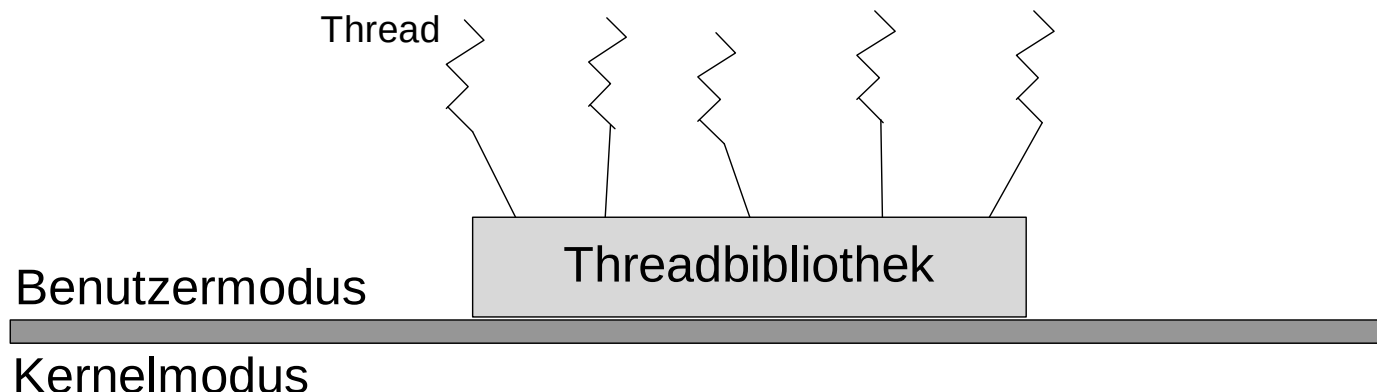


Quelle: Solomon, D. A.; Russinovich, M.: Microsoft Windows Internals, Microsoft Press, Part 1 und 2, 6. Auflage, 2013

# Implementierungsvarianten für Threads

## ■ Implementierung **auf Benutzerebene**

- auch „green threads“
- Thread-Bibliothek übernimmt das Scheduling und Dispatching für Threads
- Scheduling-Einheit ist der Prozess
- Kernel merkt nichts von Threads

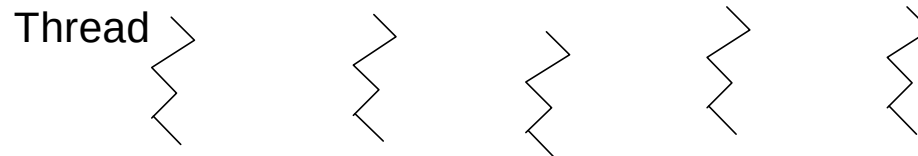


# Implementierungsvarianten für Threads

- Implementierung **auf Kernelebene**
  - auch „**red threads**“
  - Prozess ist nur noch Verwaltungseinheit für Betriebsmittel
  - Scheduling-Einheit ist hier der Thread, nicht der Prozess
  - Nicht so effizient, da Thread-Kontextwechsel über Systemcall

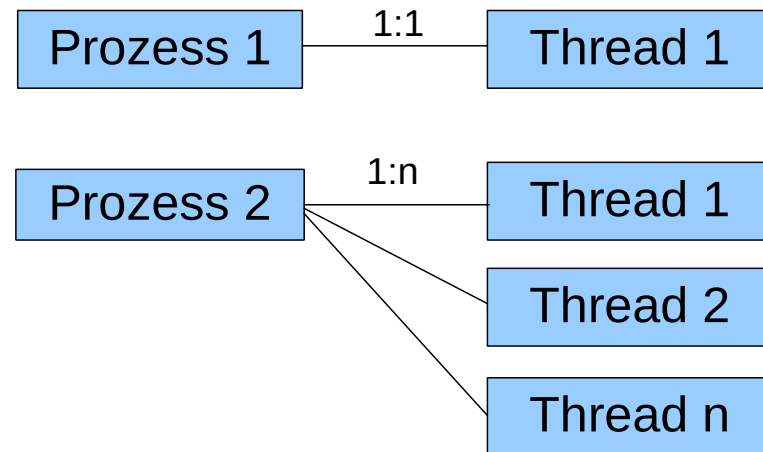
Benutzernodus

Kernelmodus



# Zuordnung von Threads zu Prozessen

- 1:1: Genau ein Thread läuft in einem Prozess
- 1:n: Mehrere Threads laufen in einem Prozess

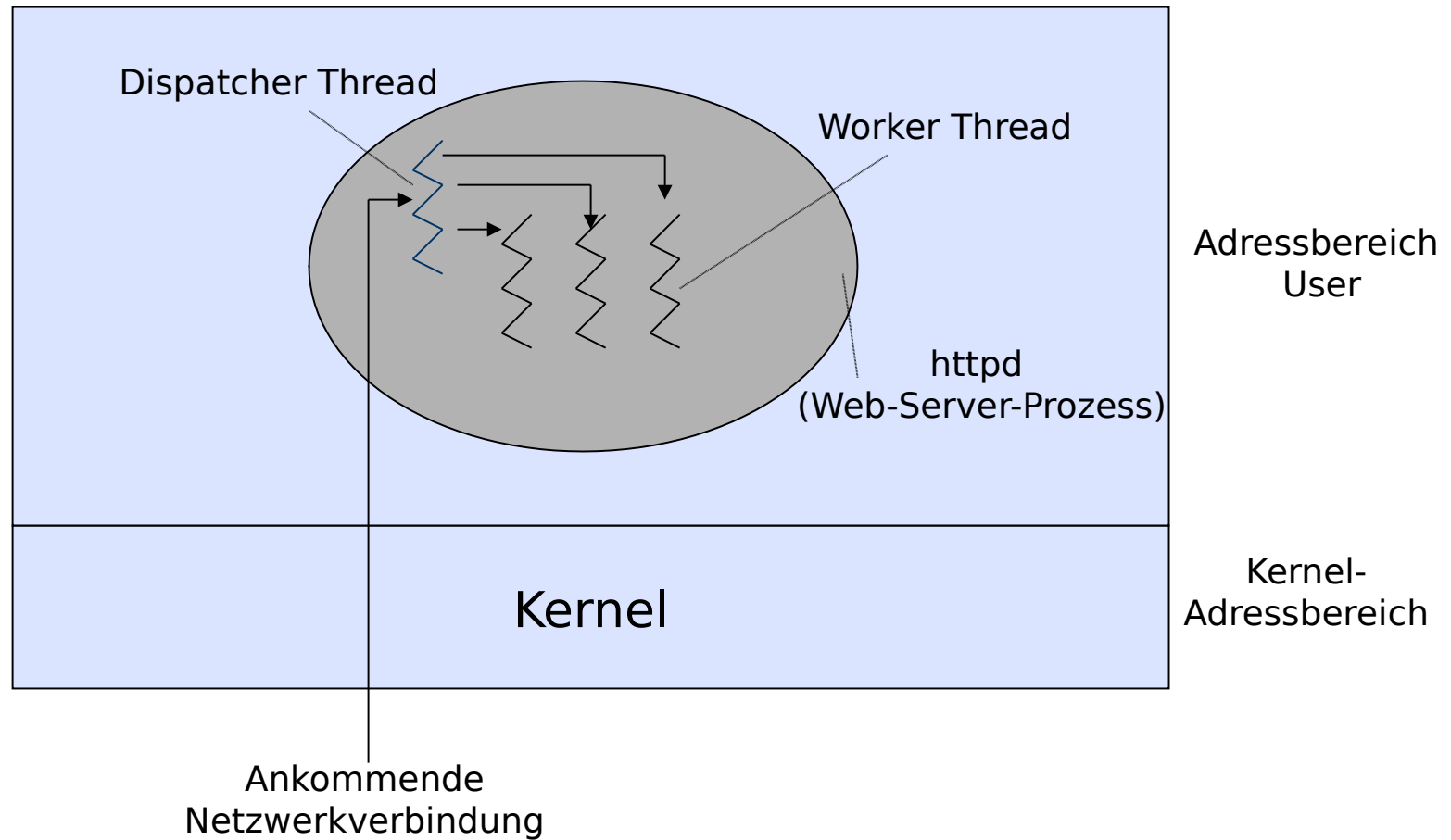


- Auch die Zuordnung von User-Level-Threads zu Kernel-Level-Threads ist wichtig
- Es muss definiert sein: Was ist die Scheduling-Einheit?

## Gründe für Threads

- Thread-Kontext-Wechsel geht **schneller** als Prozess-Kontext-Wechsel
- **Parallelisierung** der Prozessarbeit (muss aber entsprechend programmiert werden); Beispiel:
  - Ein Thread hört auf Netzwerkverbindungswünsche
  - Ein Thread führt Berechnungen durch
  - Ein Thread kümmert sich um das User-Interface (Keyboard-Eingabe, Ausgabe auf Bildschirm)
- Sinnvoll bei Systemen mit mehreren CPUs
- Einsatz z.B. im Web-Server:
  - Dispatcher-Thread wartet auf ankommende HTTP-Requests
  - Mehrere Worker-Threads bearbeiten Request

# Einsatzbeispiel für Threads: Web-Server



Quelle: *Tanenbaum, A. S.: Moderne Betriebssysteme, 3. aktualisierte Auflage, Pearson Studium, 2009*

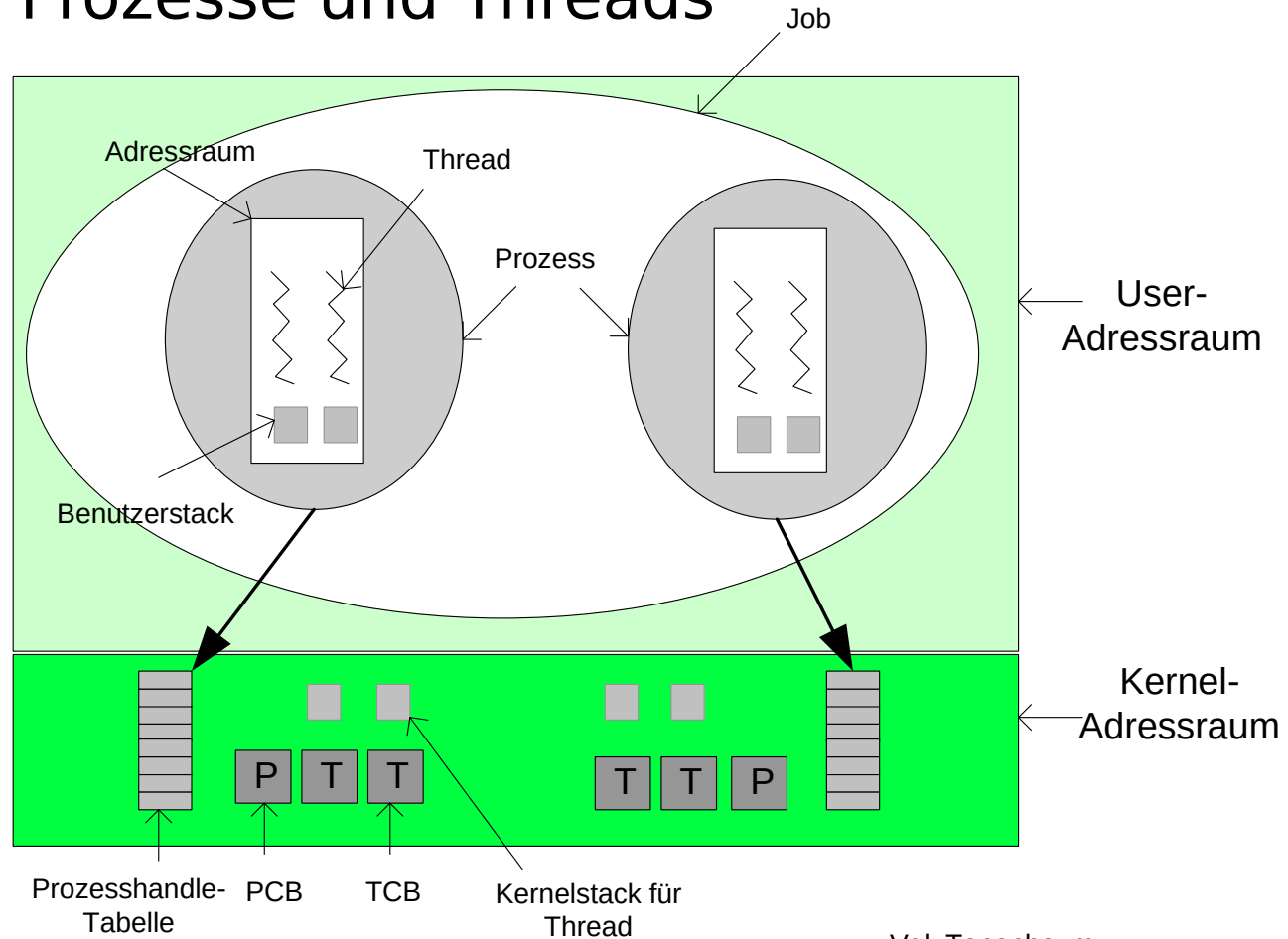
## Einsatzbeispiel für Threads: Pseudocode

```
dispatcher() {  
    while (true) {  
        r = receive_request();           // Warten auf ankommende  
        Requests                         // Request eingetroffen  
        start_thread(workerThread, r);  
    }  
}
```

```
workerThread(r) {                       // Thread zur  
    Requestbearbeitung                   // Antwort zurück an Requestor  
    a = process_request(r);  
    reply_request(a);  
}
```

# Prozess-/Thread-Verwaltung unter Windows

## ■ Jobs, Prozesse und Threads



Vgl. Tanenbaum



# Prozess-Thread-Verwaltung unter Windows

- **Job** = Gruppe von Prozessen, die als eine Einheit verwaltet werden, haben Quotas und Limits
  - Maximale Speichernutzung je Prozess
  - Maximale Anzahl an Prozessen
  - ...
- **Prozess** = Container zur Speicherung von Ressourcen
  - Threads, Speicher,...
- **Thread** = Scheduling-Einheit
- **Fiber** = Leichtgewichtiger Thread, der vom User verwaltet wird (CreateFiber, SwitchToFiber)

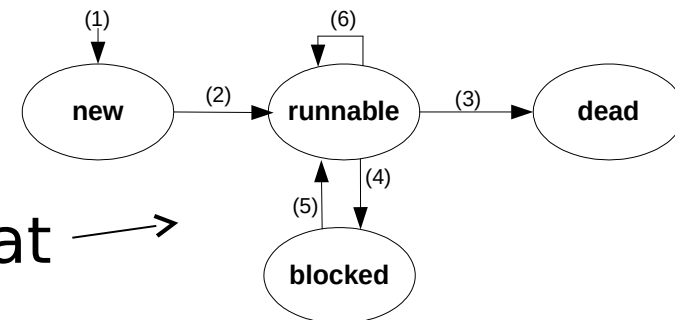
# Überblick

---

1. Prozesse und Lebenszyklus von Prozessen
2. Threads
- 3. Threads im Laufzeitsystem**

# Threads in Java, JVM und Threads

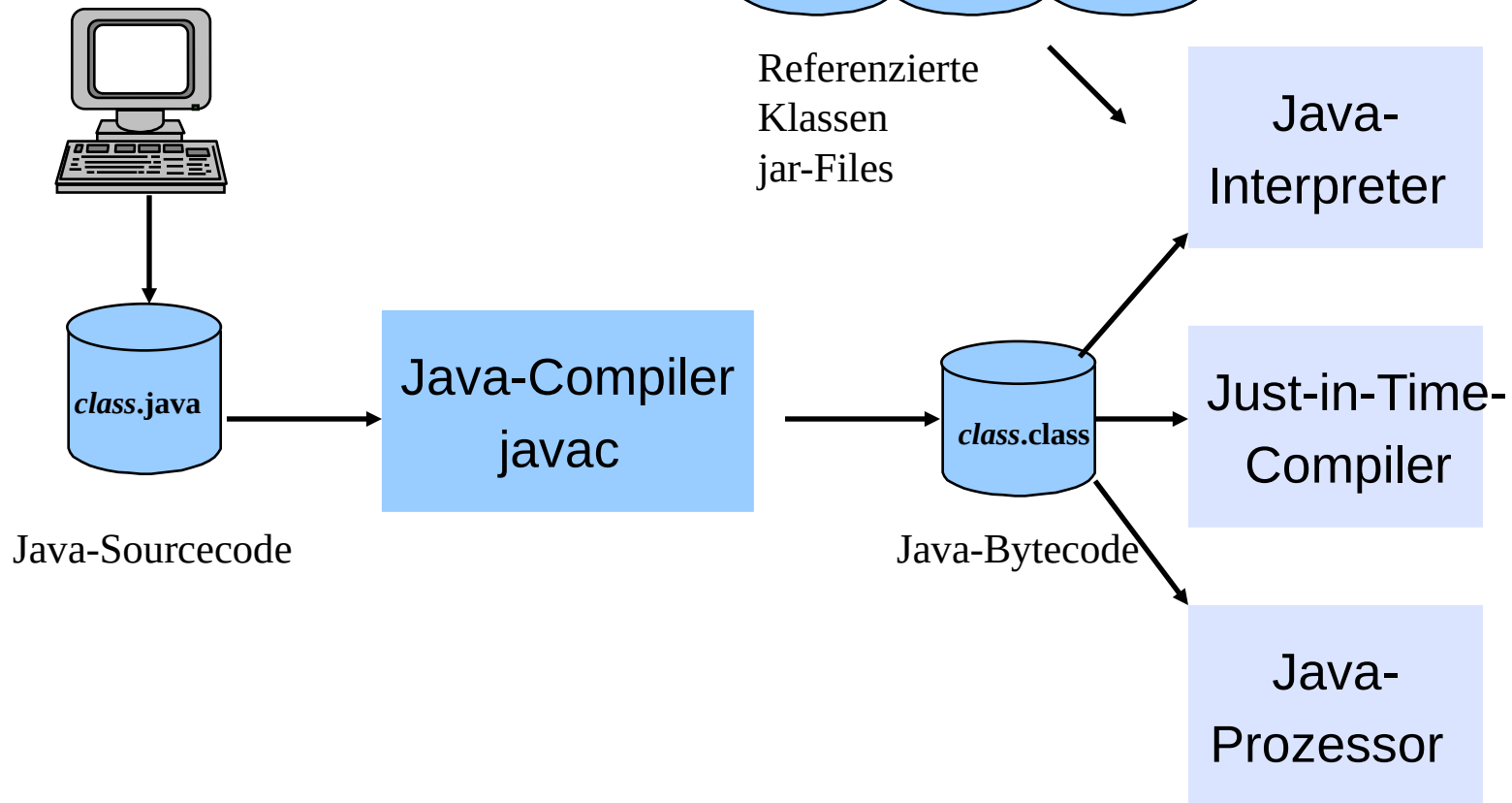
- Für jedes Programm wird eine eigene JVM gestartet
- JVM läuft in einem Betriebssystemprozess
  - Siehe z.B. im Windows Task Manager
- JVM unterstützt Threads
- Package **java.lang**
- Basisklasse **Thread**
- Vereinfachter Zustandsautomat →



- (1) Konstruktoraufwurf der Klasse Thread
- (2) Aufruf der Methode run()
- (3) Aufruf der Methode stop()
- (4) Aufruf der Methode sleep()
- (5) Aufruf der Methode resume()
- (6) Aufruf der Methode yield()

# Einschub: Übersetzungsvorgang und Ablauf eines Java-Programms

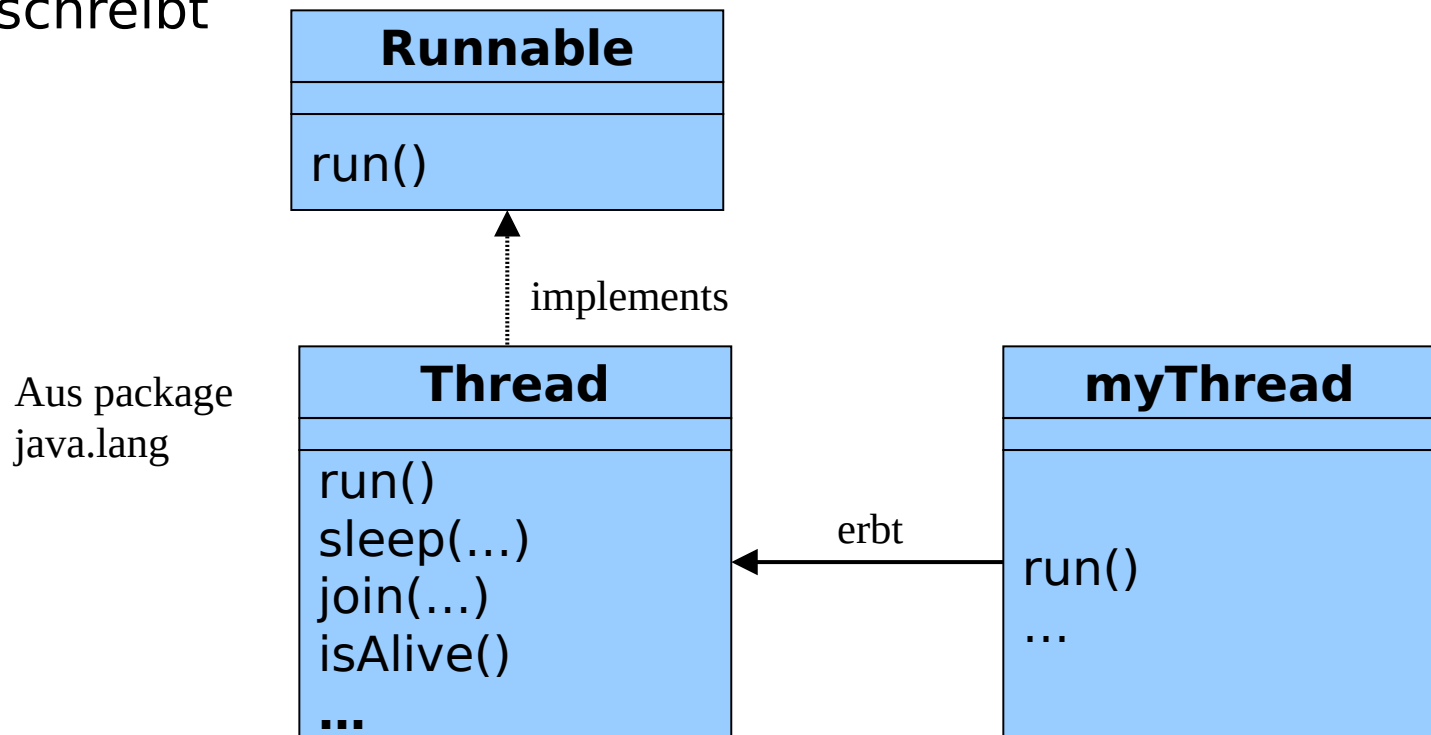
Entwicklungsumgebung:  
Eclipse, ...



# Threads in Java

## Die Klasse Thread und das Interface Runnable

- Nebenläufigkeit wird durch die Klasse *Thread* aus Package `java.lang` unterstützt
- Eigene Klasse definieren, die von *Thread* abgeleitet ist und die Methode `run()` aus Interface *Runnable* überschreibt



# Threads in Java

## Beispiel: Eine einfache Thread-Klasse ...

```
import java.lang.Thread;

class myThread extends Thread { // meine Thread-Klasse

    String messageText;

    public myThread(String messageText)
    {
        this.messageText = messageText;
    }

    public void run()           // Methode, welche die eigentliche Aktion
                                // ausführt, definiert in Interface Runnable
    {
        for (;;) {
            System.out.println("Thread " + getName() + ": " + messageText);
            try {
                sleep(2000);
            }
            catch (Exception e) { /* Exception behandeln */ }
        }
    }
}
```

# Threads in Java

## Beispiel: ... und deren Nutzung

```
public class myThreadTest {  
  
    public static void main(String args[])  
    {  
        myThread t1;  
        t1 = new myThread("...auf und nieder immer wieder...");  
        t1.start();  
        if (t1.isAlive()) {  
            for (int i=0; i < 100000000; i++) {} // nonsense  
            try {  
                t1.join(10000);  
            } catch (InterruptedException e) { /* Exception behandeln */ }  
  
            System.out.println("Mainprogramm stoppt Thread myThread!!!");  
            t1.stop(); // deprecated  
            System.out.println("Thread " + t1.getName() + " beendet");  
        }  
    }  
}
```

- Was passiert in diesem Programm?

# Threads in Java

## Beispiel: Erläuterungen

- Innerhalb der Methode *start()* wird automatisch die *run()*-Methode des *Runnable*-Objekts aufgerufen
- Die Methode *join()* ohne Parameter wartet bis der Thread „stirbt“, *join(long millis)* wartet „millis“ Millisekunden und dann wird weiter gemacht
- Weitere Methoden der Klasse Thread:
  - *getPriority()*: Thread-Priorität ermitteln
  - *isAlive()*: Prüfen, ob Thread lebt
  - *getThreadGroup()*: Thread-Gruppe des Threads ermitteln
  - *interrupt()*: Thread unterbrechen
  - *getName()*: Thread-Namen ermitteln
  - ...
  - Mehrere Konstruktoren



# Einschub: System-Threads

---

- Threads sind in Java als Gruppen hierarchisch organisiert:
  - Thread-Gruppe **system** für die Threads des Systems (der JVM)
  - Thread-Gruppe **main** für die benutzerspezifischen Threads als Untergruppe von system
- Threads der Gruppe **system**:
  - **Finalizer**: Ruft für freizugebende Objekte die finalizer-Methode auf
  - ...
  - Signal dispatcher

# Einschub: System-Threads

---

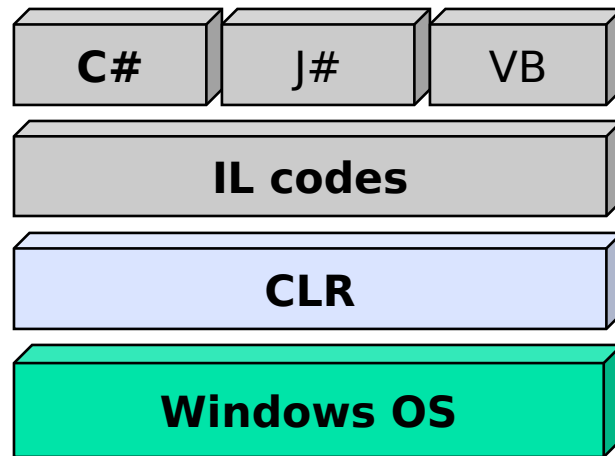
- Weitere Threads:
  - **Garbage Collection**: hat sehr niedrige Priorität (niedriger als Idle-Thread, wartet auf Signal von Idle-Thread)
  - **Idle**: Wenn er läuft, setzt er ein Kennzeichen, das der **Garbage Collection** Thread als Startsignal betrachtet, um etwas zu tun
    - **Idle** wird nur aufgerufen, wenn die JVM sonst nichts zu tun hat

# C# Ausflug: .NET Framework: CIL, CLR, FCL

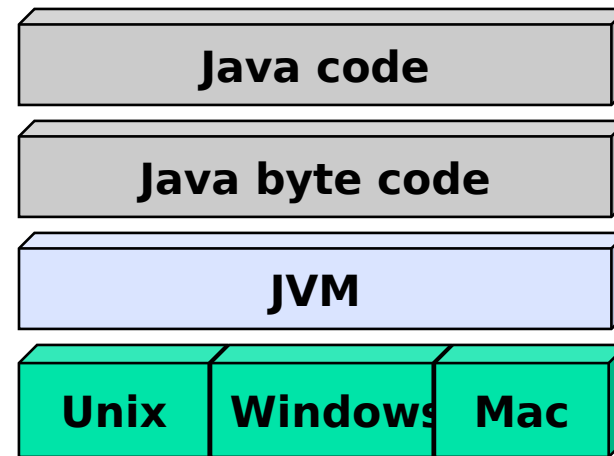
---

- .NET Framework: Plattform zur Entwicklung und Ausführung von Anwendungsprogrammen
- CIL = Common Intermediate Language ist ein Zwischencode
  - entspricht Java Byte Code
- CLR = Common Language Runtime
  - entspricht JVM
- Alle Microsoft-Compiler erzeugen CIL-Code
- FCL = Framework Class Library
  - Klassenbibliothek mit vielen Basisklassen
  - in Namespaces geordnet

# C# Ausflug: CLR versus JVM



.NET - Lösung



Java - Lösung

IL = Intermediate Language

# C# Ausflug: Assembly

- Grundbausteine für Weitergabe, Versionskontrolle, Wiederverwendung, Sicherheitsberechtigungen
- Mehrere Quelldateien ergeben zusammen nach der Ausführung eine ausführbare Datei (Assembly)
- Dateinamen .dll und .exe, unterscheiden sich aber kaum voneinander
  - exe-Dateien haben konkreten Startpunkt (main())
  - dll-Dateien benötigen eine exe-Datei als Host
- Assembly enthält Metadaten (Manifest)
  - Objektname, Attribute,...

# C# Ausflug: Threads in C#

---

- optional: Threads in C#: siehe Folien  
optional/05-2\_Prozesse\_und\_Threads\_Csharp.odp

# Gesamtüberblick

---

- ✓ Einführung in Computersysteme
- ✓ Entwicklung von Betriebssystemen
- ✓ Architekturansätze
- ✓ Interruptverarbeitung in Betriebssystemen
- ✓ **Prozesse und Threads**
- 5. CPU-Scheduling
- 6. Synchronisation und Kommunikation
- 7. Speicherverwaltung
- 8. Geräte- und Dateiverwaltung
- 9. Betriebssystemvirtualisierung