DUALE HOCHSCHULE SH

Prozesse und Threads

Computing-Plattformen und Netzwerke

auf Basis der Unterlagen von
Prof. P. Mandl (HS München)

Prozesse

Definition: Prozesse

- Informelle Definitionsansätze: Ein **Prozess** (manchmal auch Task genannt)
  - ist die Ausführung (Instanzierung) eines Programms auf einem Prozessor
  - ist eine dynamische Folge von Aktionen verbunden mit entsprechenden Zustandsänderungen
  - ist die gesamte Zustandsinformation der Betriebsmittel eines Programms
  - Die Ablaufumgebung für ein Programm, die das Betriebssystem bereitstellt

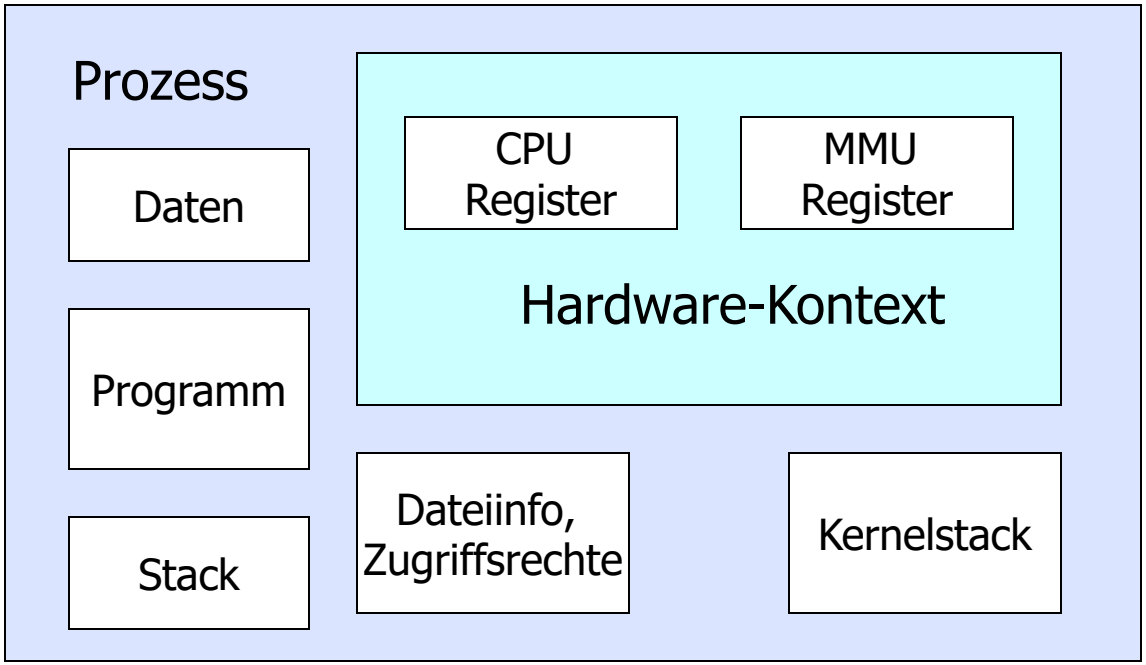
Virtuelle Prozessoren

- Das Betriebssystem ordnet im Multitasking jedem Prozess einen **virtuellen Prozessor** zu
- Echte Parallelarbeit, falls mehreren virtuellen Prozessoren jeweils 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 Kontext-Switch)
    - bisheriger Prozess wird gestoppt
    - neuer Prozess (re)aktiviert

Prozesskontext



MMU = Memory Management Unit

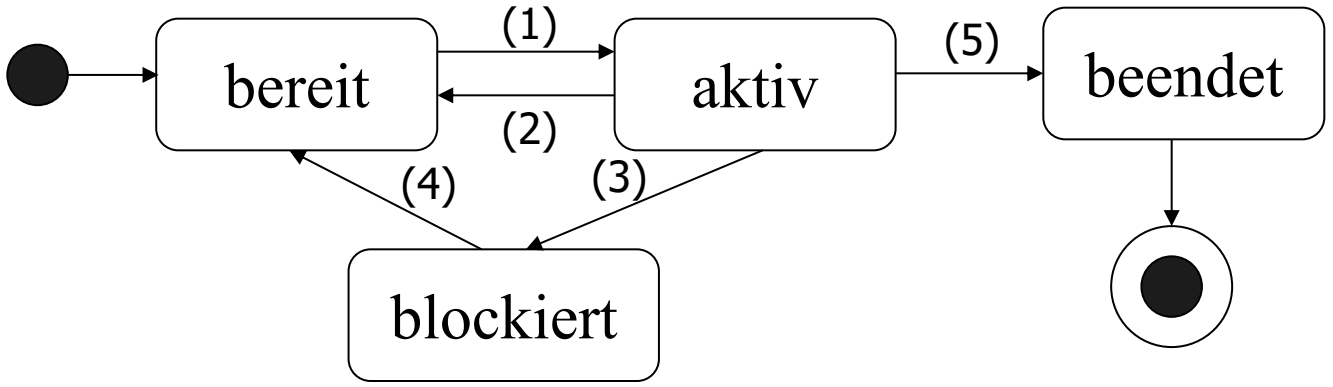
- Prozesskontext = gesamte Zustandsinformation zu einem Prozess
- Kernelstack = Stack für Systemaufrufe des Prozesses

Prozesslebenszyklus

- Ein Prozess wird mit Mitteln des Betriebssystems erzeugt, Beispiel in Unix: Systemaufruf ***fork()***
  - Hauptspeicherbereich und weitere Ressourcen zuordnen
  - Programmcode und Daten in Speicher laden
  - Prozesskontext laden und Prozess starten, wenn CPU verfügbar ist
- Für das Beenden eines Prozesses gibt es mehrere Gründe:
  - Normaler exit
  - Error exit (vom Programmierer gewünscht, fatal error)
  - Durch Kernel 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, die 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 und enthält u.a.:
  - Programmzähler
  - Prozesszustand
  - Priorität
  - Verbrauchte Prozessorzeit seit dem Start des Prozesses
  - Prozessnummer (PID), Elternprozess (PID)
  - Zugeordnete Betriebsmittel, z.B. Dateien (Dateideskriptoren)
  - …
- Meist ist der PCB auf mehrere Datenstrukturen verteilt

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** oder **systemd** (PID 1), bei macOS heißt der Prozess **launchd**
    - Urvater aller weiteren Prozesse

Prozessverwaltung unter Unix: Prozesserzeugung - fork

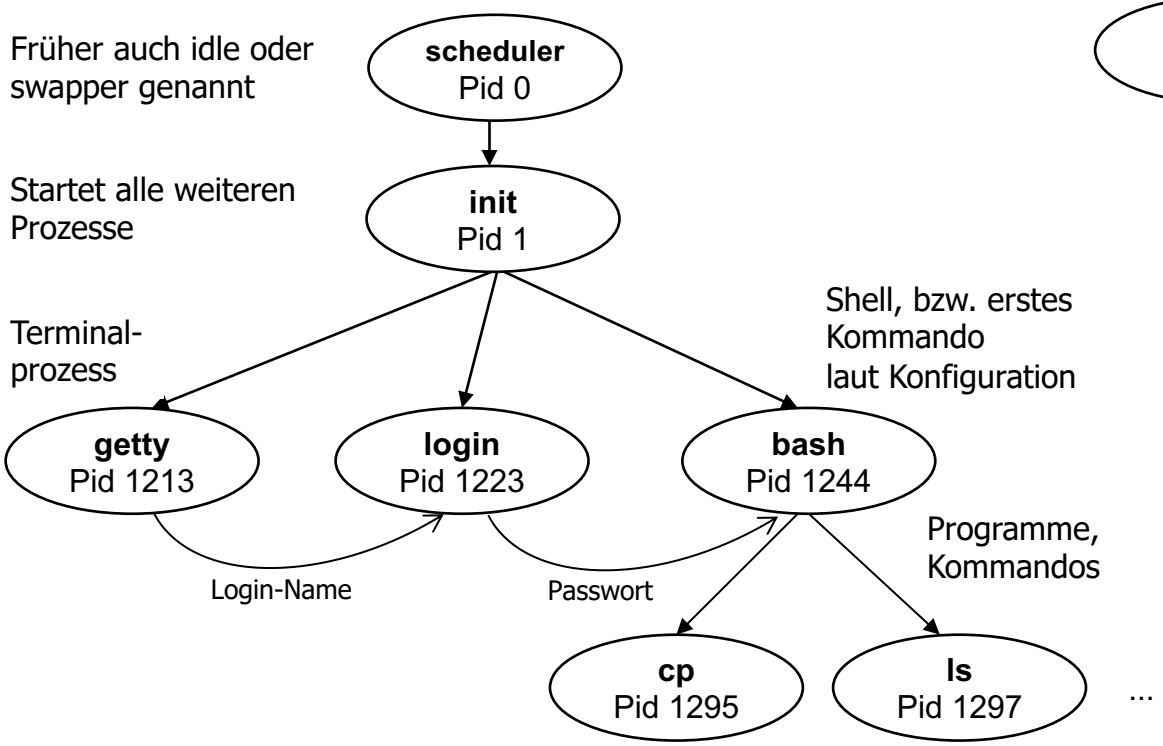
- Ein Prozess wird unter Unix durch einen ***fork()-*** Aufruf des Vaters erzeugt
- Der Kindprozess erbt dessen Umgebung **als Kopie**:
  - Alle offenen Dateien und Netzwerkverbindungen
  - Umgebungsvariablen
  - Aktuelles Arbeitsverzeichnis
  - Datenbereiche
  - Codebereiche
- Durch den System-Call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

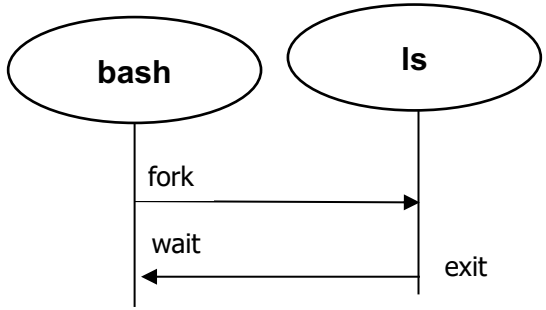
Prozesserzeugung unter Unix (C-Beispiel)

```
01: int main()
02: {
03:    int ret;          // Returncode von fork
04:    int status;       // Status des Kindprozesses
05:    pid_t pid;        // pid_t ist ein spezieller Datentyp, der
06:                      // eine PID beschreibt
07:    ret = fork();     // Kindprozesses wird erzeugt
08:    if (ret == 0) {
09:      // Anweisungen, die im Kindprozess ausgeführt werden
         ...
10:      exit(0); // Beenden des Kindprozesses mit Status 0 (ok)
11:    }
12:    else {
13:       // Anweisungen, die nur im Elternprozess ausgeführt werden
14:       // sollen
15:       // Zur Ablaufzeit kommt hier nur der Elternprozess rein
16:       // (Returncode = PID des Kindprozesses)
          ...
17:       pid = wait(&status);          // Warten auf das Ende des
18:                                     // Kindprozesses
19:       exit(0); // Beenden des Vaterprozesses mit Status 0 (ok)
20:    }
21: }
```

Unix-Prozessbaum

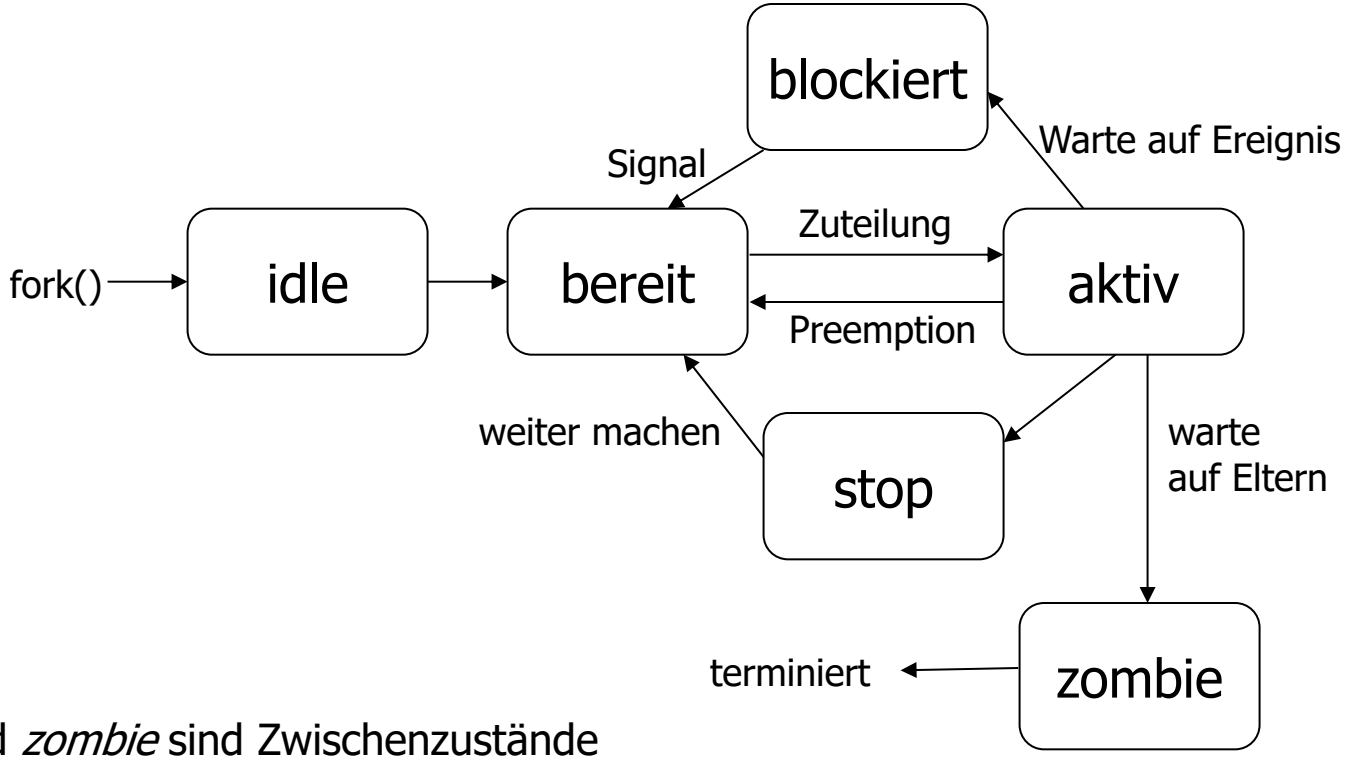
- Je Terminal wartet ein getty-Prozess auf eine Eingabe (Login)
- Nach erfolgreichem Login wird ein Shell-Prozess eröffnet
- Jedes Kommando wird gewöhnlich in einem eigenen Prozess ausgeführt





Zustandsautomat eines Unix-Prozesses, allgemein

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



idle und *zombie* sind Zwischenzustände

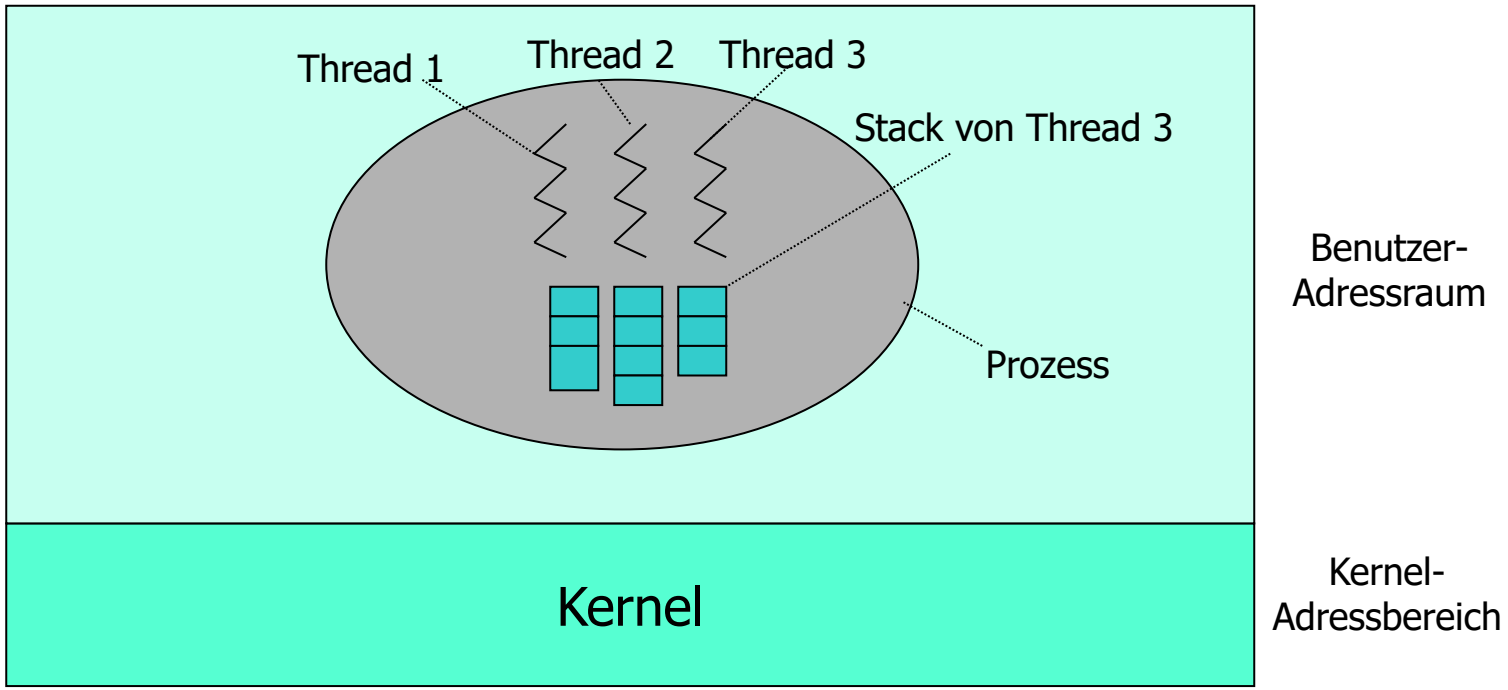
Threads

Definition: Threads

- Threads sind **leichtgewichtige** Prozesse (lightweight processes, LWP)
- Threads teilen sich Ressourcen im Prozess:
  - **Gemeinsamer Adressraum**
  - Offene Files, Netzwerkverbindungen **...**
- Eigener **Zustandsautomat** ähnlich wie Prozess
- Mehrere Threads im Prozess → **Multithreading**
- Threads sind nicht gegeneinander geschützt
  - Synchronisationsmaßnahmen erforderlich

Threads, Stack

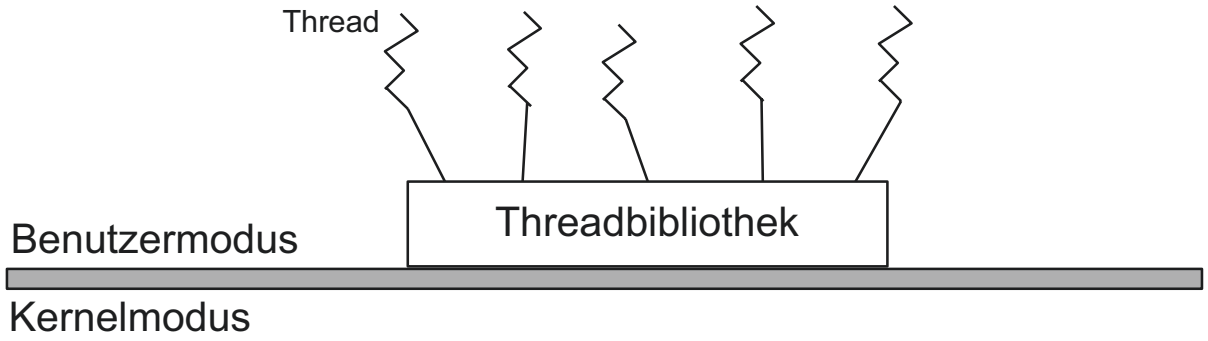
- Threads haben einen eigenen log. Programmzähler, einen eigenen log. Registersatz und einen eigenen Stack



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

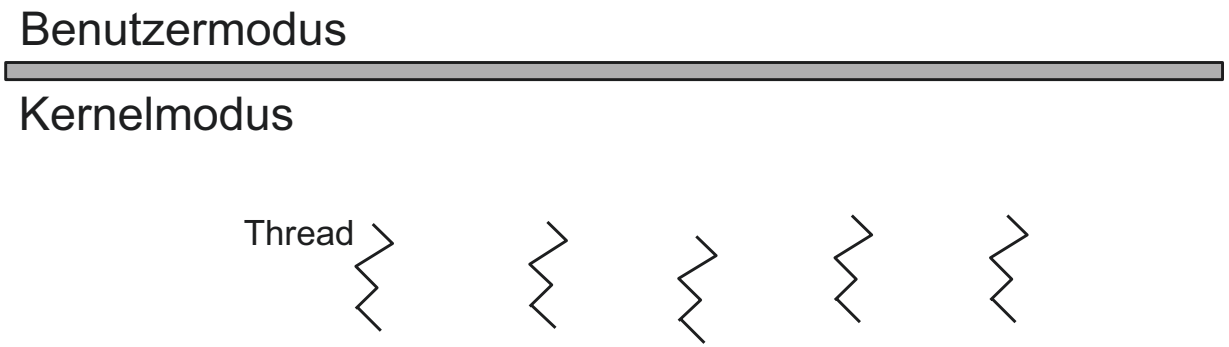
Implementierungsvarianten für Threads

- Implementierung **auf Benutzerebene**
  - 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**
  - Prozess ist nur noch Verwaltungseinheit für Betriebsmittel
  - Scheduling-Einheit ist hier der Thread, nicht der Prozess
  - Nicht so effizient, da Thread-Kontextwechsel über Systemcall ausgeführt wird



Zuordnung von Threads zu Prozessen

- 1:1: Genau ein Thread läuft in einem Prozess (früher in Unix)
- 1:n: Mehrere Threads laufen in einem Prozess (heute in gängigen Universalbetriebssystemen üblich)

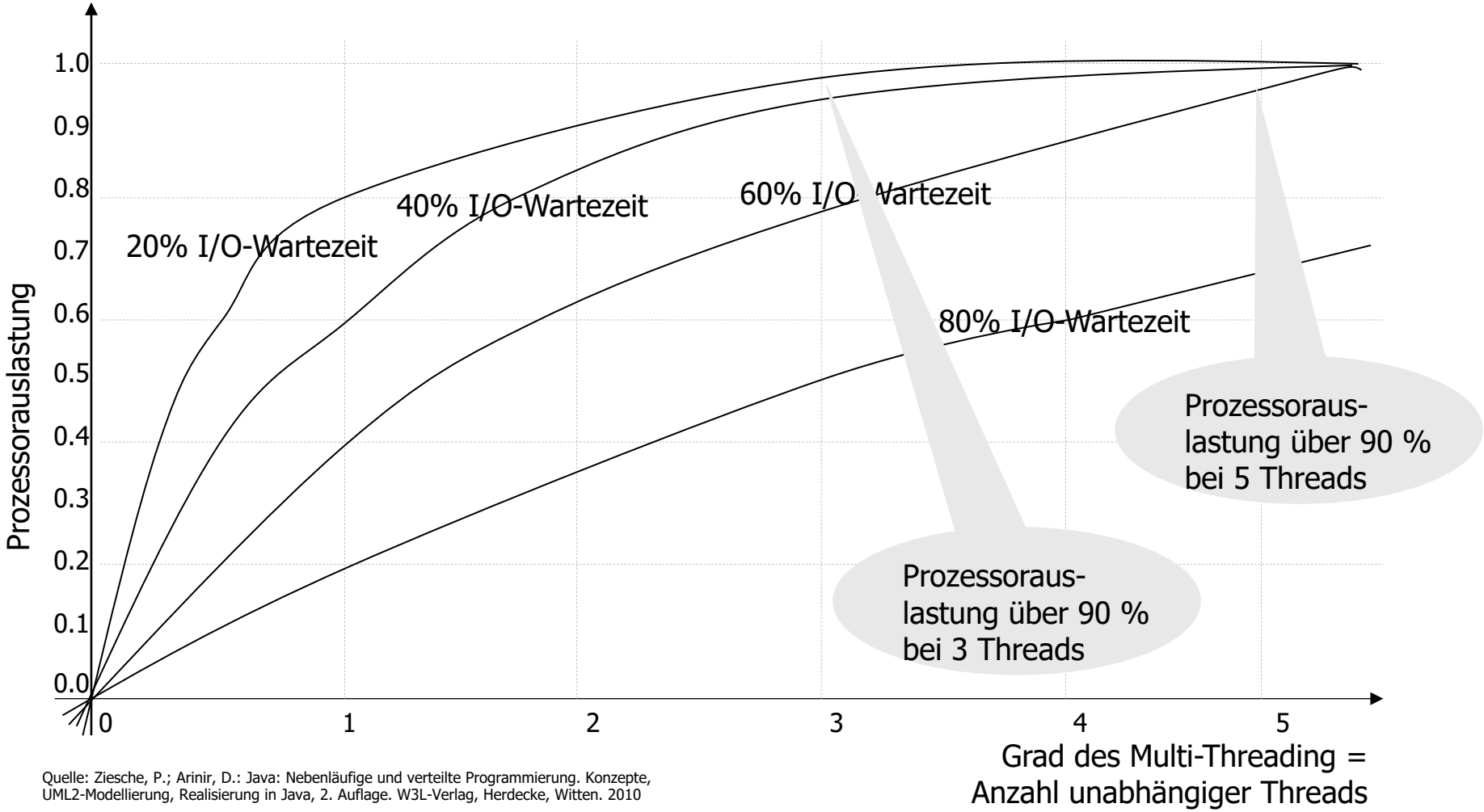
- Es muss definiert sein: Was ist die Scheduling-Einheit?

Gründe für Threads (1)

- 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 vor allem bei Systemen mit mehreren CPUs

Gründe für Threads (2)

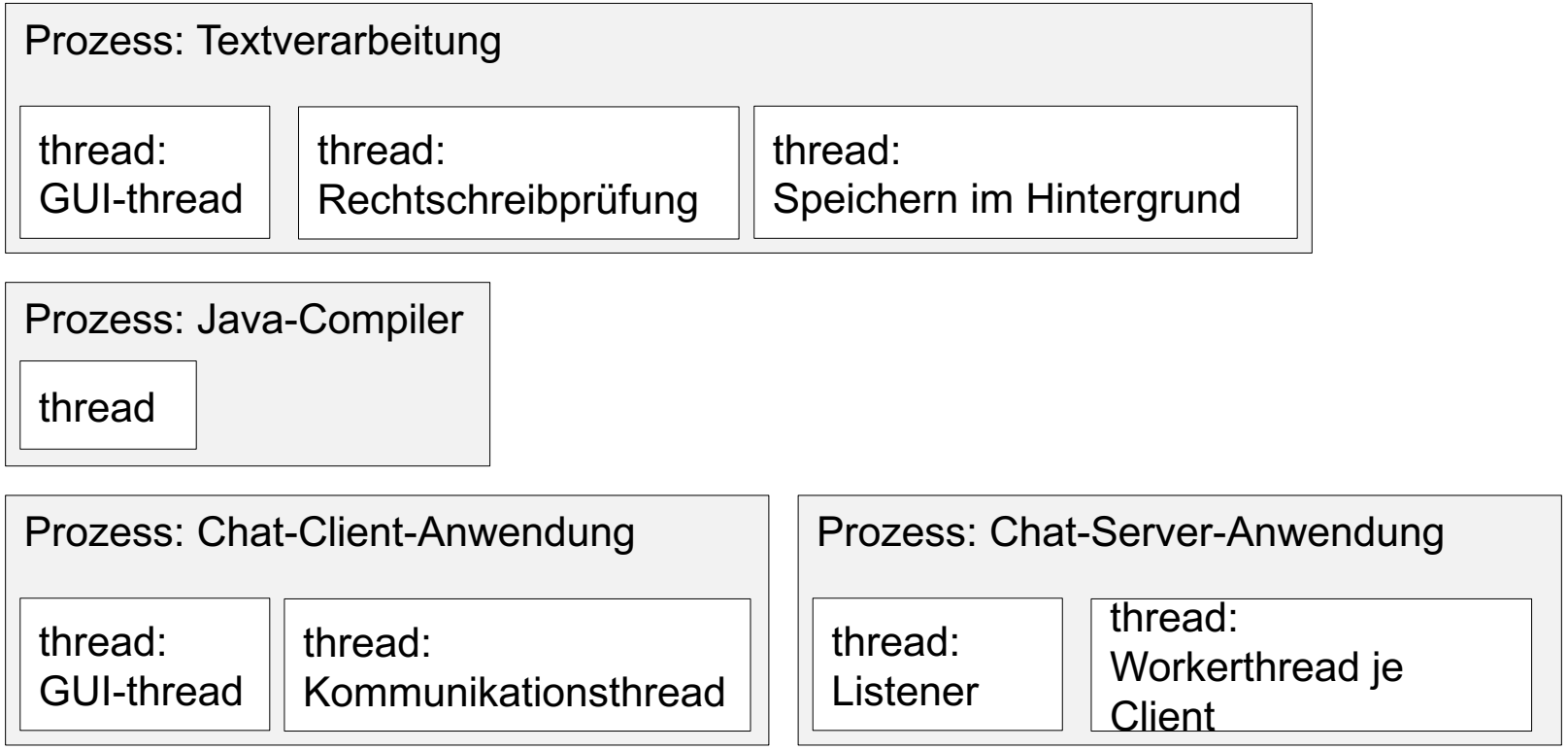
- Prozessorauslastung in Bezug zum Grad des Multi-Threading



Quelle: Ziesche, P.; Arinir, D.: Java: Nebenläufige und verteilte Programmierung. Konzepte, UML2-Modellierung, Realisierung in Java, 2. Auflage. W3L-Verlag, Herdecke, Witten. 2010

Einsatzbeispiel für Threads: Textverarbeitung, Compiler, Chat

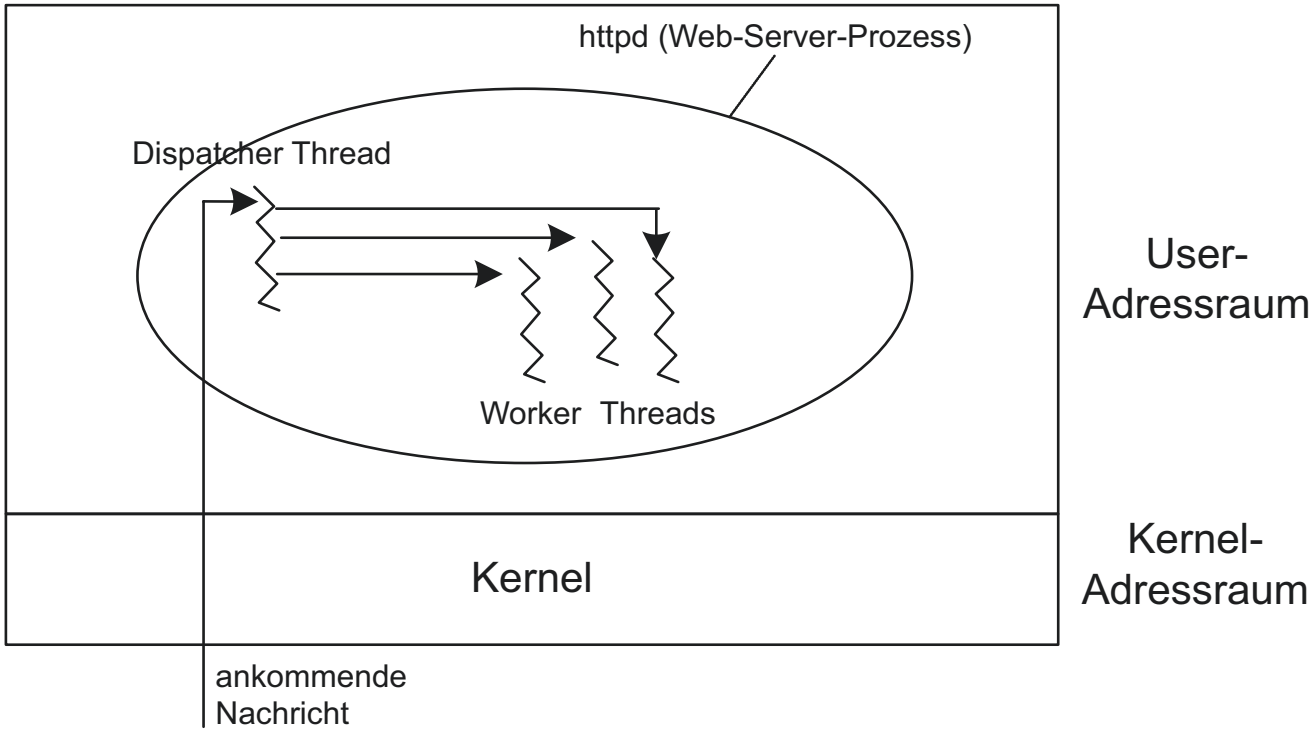
- Typische Thread-Nutzung für verschiedene Anwendungen



Quelle: Ziesche, P.; Arinir, D.: Java: Nebenläufige und verteilte Programmierung. Konzepte, UML2-Modellierung, Realisierung in Java, 2. Auflage. W3L-Verlag, Herdecke, Witten. 2010

Einsatzbeispiel für Threads: Web-Server

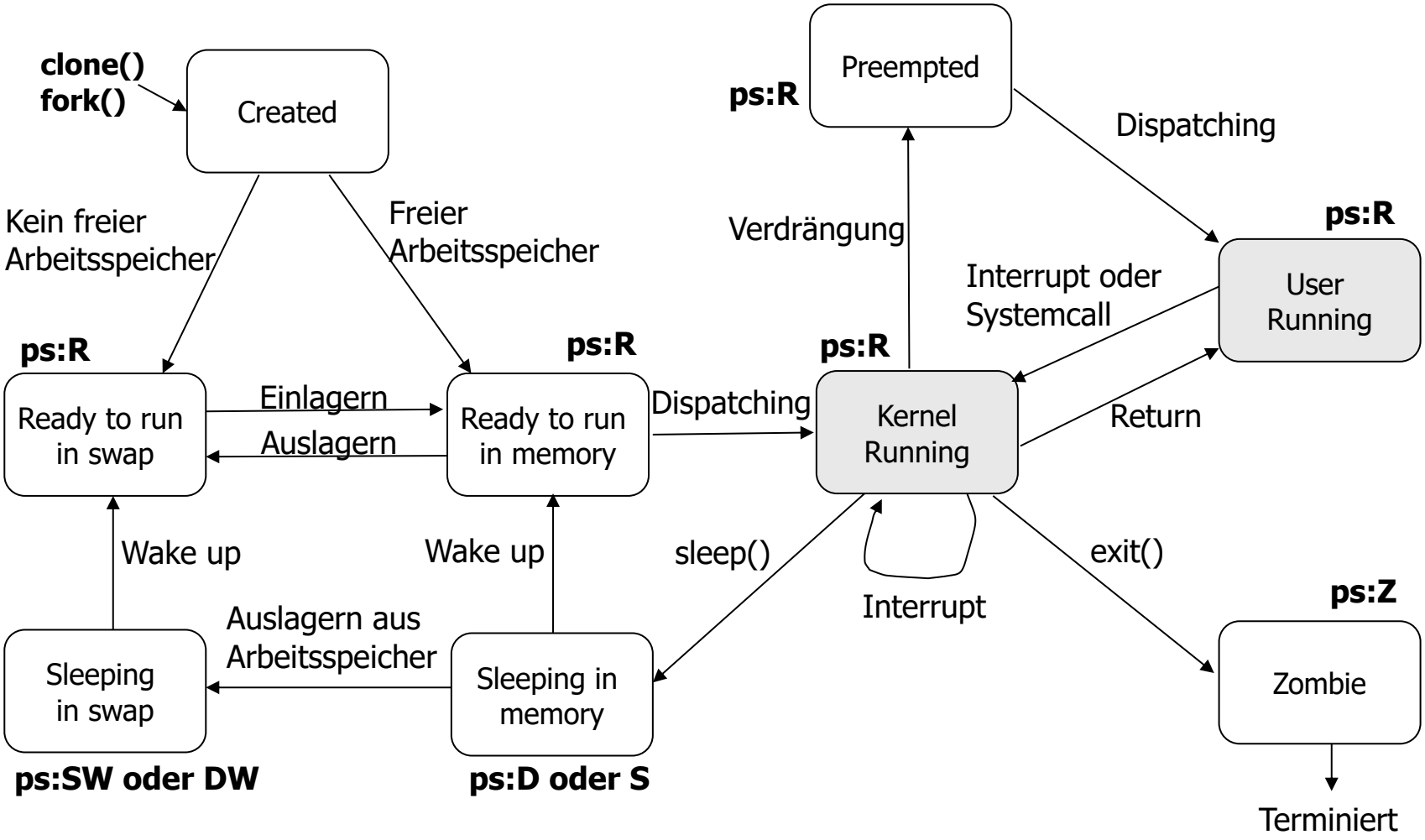
- Einsatz z.B. im Web-Server:
  - Dispatcher-Thread wartet auf ankommende HTTP-Requests
  - Mehrere Worker-Threads bearbeiten Request



Einsatzbeispiel für Threads: Pseudocode

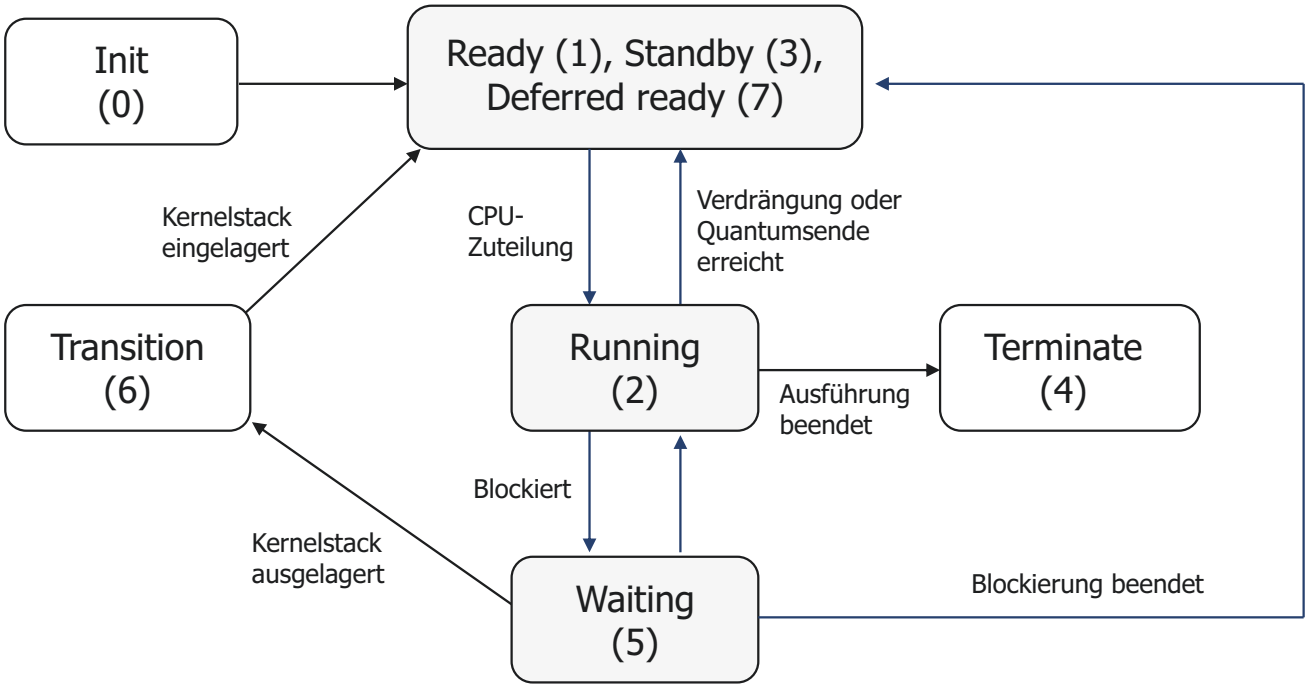
```
01: Dispatcher() {
02:       while (true){ // Warten auf ankommende Requests
03:             r= receive_request();
                // Request eingetroffen, wird bearbeitet
04:             start_thread(workerThread, r);
05:       }
06: }

07: workerThread(r) {    // Thread zur Requestbearbeitung
08:             a = process_request(r);
                // Antwort zurück an Requestor
09:             reply_request(a);
10: }
```

Zustandsautomat eines Linux-Prozesses = Thread



Prozess-/Thread-Verwaltung unter Windows

- Jobs, Prozesse und Threads

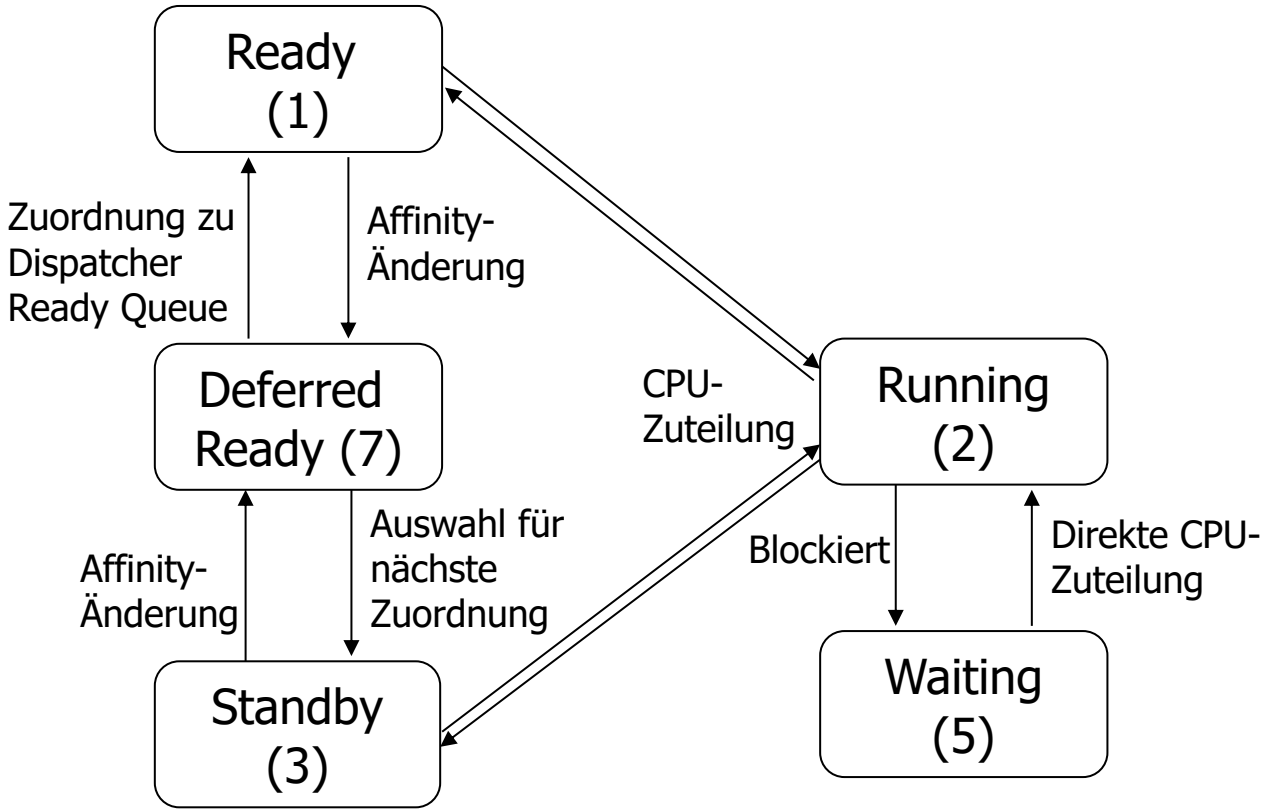


Thread-Zustandsautomat unter Windows: Spezielle Zustände

- Ready
  - Thread wartet auf Ausführung
- Deferred Ready
  - Thread wurde für einen Prozessor ausgewählt
  - Noch nicht auf dem Prozessor abgelaufen
  - Wartezustände in Lockzuständen werden minimiert (Locks werden später behandelt)
- Standby
  - Thread wurde für einen speziellen Prozessor ausgewählt und kommt als nächstes dran (in den Zustand Running)
  - Nur ein Thread je Prozessor in diesem Zustand
  - Verdrängung möglich, wenn Thread mit höherer Priorität „runnable" wird

Thread-Zustandsautomat unter Windows: Spezielle Zustandsübergänge

- Zustände Ready, Standby und Deferred ready zusammengefasst

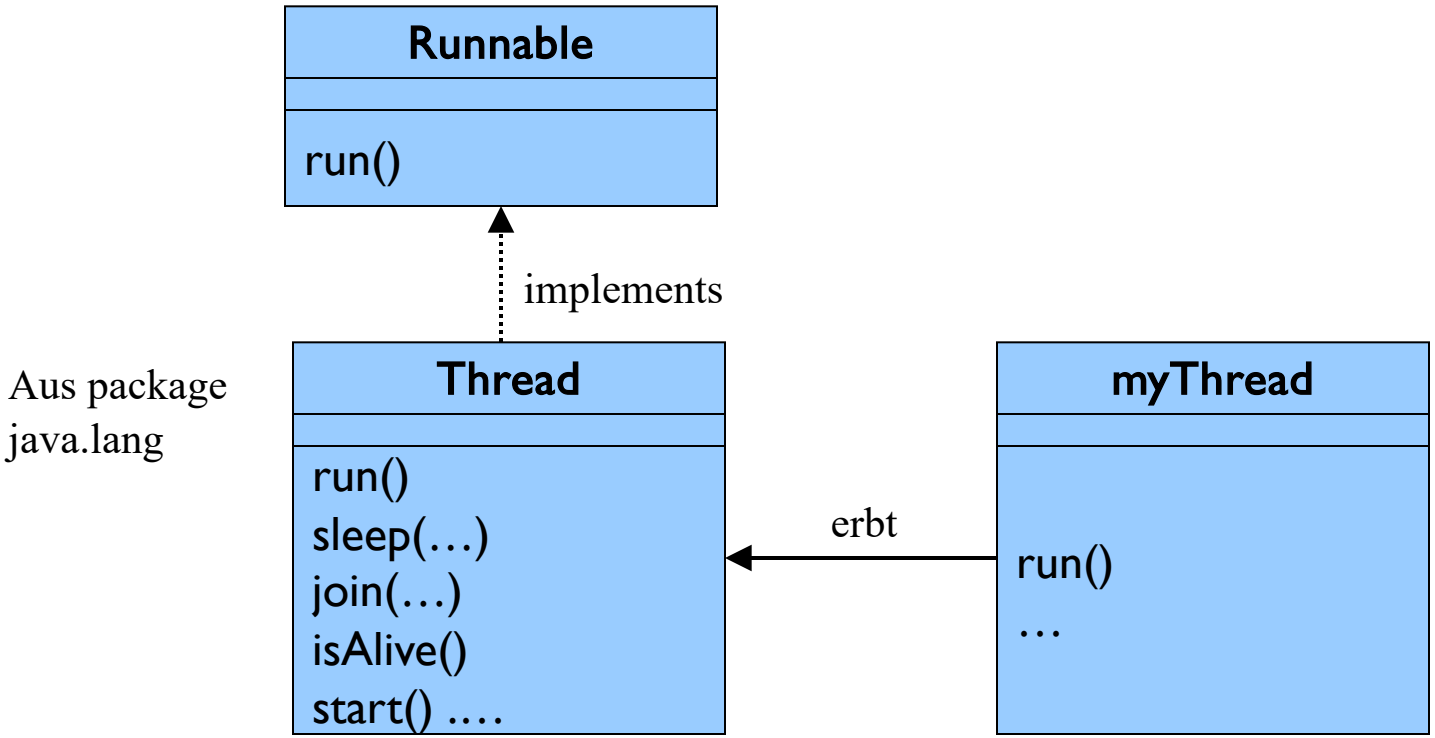


Threads in Java
JVM und Threads

- Für jedes Programm wird eine eigene JVM gestartet
- JVM läuft in einem Betriebssystemprozess
  - Siehe z.B. im Windows Task Manager
- JVM unterstützt Threads

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



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