Kapitel VIII

Threads

Motivation (I)

- Bisher:

  Betrachtung von Prozessen, wobei ein Prozess einen eigenen Adressraum und einen einzigen Ausführungsfaden (sog. Thread of Control) aufweist.

- Jedoch:

  Moderne Applikationen erfordern innerhalb des Prozess-Kontexts (quasi-) parallele Verarbeitung.

  In modernen Betriebssystemen können deshalb innerhalb eines Prozesskontexts Aufgaben parallel verarbeitet werden.

- Threads liefern den hierfür notwendigen Mechanismus und stellen eine Art „leichtgewichtige Prozesse“ dar.

Beispiel 6.1: **Anwendungsbeispiel für Threads**

Es soll eine Anwendung mit grafischer Benutzeroberfläche entwickelt werden,

in der eine aufwändige Berechnung mit Fortschrittsbalken angezeigt wird:

- Eine solche Anwendung verfügt mindestens über einen UI-Thread für die Verwaltung von Events und Benutzereingaben, sowie Ausführung von Applikationscode
- Wird nun im Hintergrund eine aufwändige Berechnung durchgeführt, wird der UI-Thread für diese Berechnung benötigt → Benutzeroberfläche friert ein

→ Lösung: aufwändige Berechnung in einem weiteren Thread auszuführen,

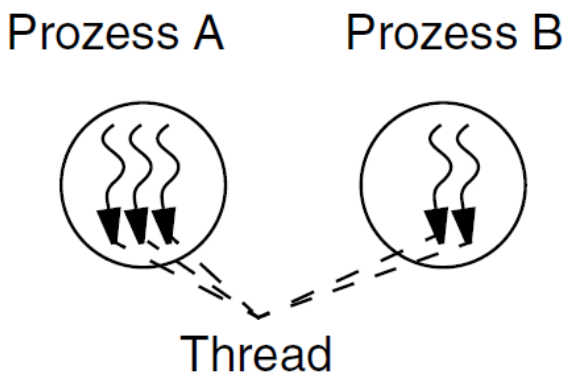
sodass der UI-Thread während der Berechnung nicht blockiert

Grundlagen (III)

- Ein Thread stellt eine parallel verarbeitbare Aktivität innerhalb eines Prozesses dar
- Einen Thread kann man sich als einen kleinen Prozess innerhalb eines Prozesses bzw. als leichtgewichtigen Prozess vorstellen
- Threads werden häufig auch als Stränge, (Ausführungs-) Fäden oder Leichtgewichtsprozesse (LWP) bezeichnet
- Bisher: Quasi-parallele Verarbeitung durch schnelles hin- und herschalten zwischen Prozessen
- Jetzt: Ergänzung durch Threads, mit deren Hilfe innerhalb eines Prozesskontexts hin und her geschaltet werden kann

Grundlagen (IV)

- Vorteile bei der Arbeit mit Threads:
  - Threads sind schneller erzeugt bzw. zerstört als Prozesse: I.d.R.: Faktor 10 – 100 schneller
  - Threads teilen sich im Gegensatz zu Prozessen den Adressraum im Hauptspeicher
  - Threads ermöglichen echte Parallelität innerhalb einer Anwendung
- Grundlage für die Arbeit mit Threads ist das Thread-Modell:



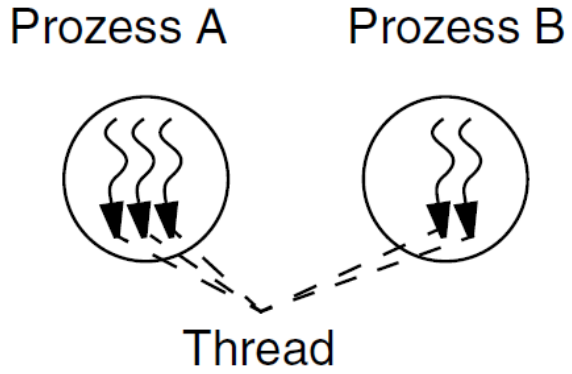
Quelle: *[BS15]*

Definition 6.1: **Thread**

Ein Thread ist ein Ausführungsfaden innerhalb eines Prozesses, welcher parallel ablaufende Aktivitäten innerhalb ein und derselben Applikation erlaubt. Die Threads innerhalb eines Prozesses nutzen einen gemeinsamen Adressraum sowie gemeinsame Ressourcen.

Einordnung von Threads im Prozess-Modell (I)

- Threads spielen untergeordnete Rolle im Prozessmodell:
  - Prozess stellt übergeordneten „Container“ als Ausführungsumgebung für Threads dar
  - Prozess stellt Ressourcen für Threads zur Verfügung und verwaltet diese
  - Threads nutzen die Ressourcen innerhalb einer Prozessumgebung gemeinsam



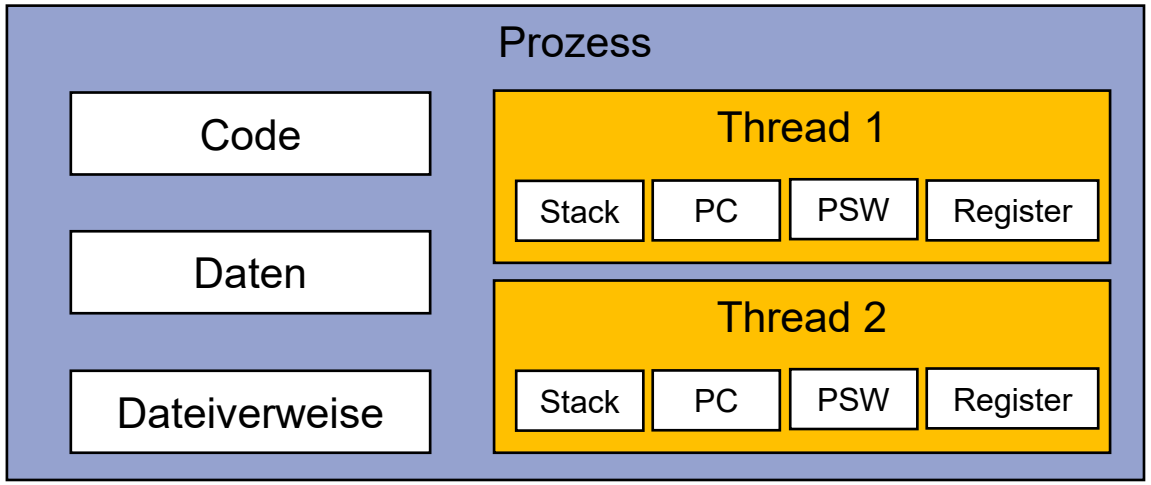
- Innerhalb eines Prozesses besteht zwischen Threads kein Schutz gegen Einsicht und Veränderung von Prozess-Daten
- Folglich: Betriebssysteme müssen neben dem Thread-Modell auch das Prozess-Modell unterstützen, ansonsten keine Kontrolle bezüglich Einsicht und Veränderung von Daten

Quelle: *[BS15]*

Einordnung von Threads im Prozess-Modell (II)

- Der umgebende Prozess besitzt die Ressourcen, wohingegen der Thread den Code ausführt
- Dabei gilt:

  Ein neu erzeugter Prozess besteht aus mindestens einem Thread, in dem der Code ausgeführt wird
- Threads besitzen nur Teilmenge der Ressourcen eines Prozesses selbst und teilen sich Code, Daten und Dateien:



Prozesse vs. Threads (I)

- Prozesse und Threads unterscheiden sich in der Art der Parallelität:

  - **Prozess-Parallelität:**

    Mehrere voneinander isolierte Benutzerprogramme werden parallel auf einem Rechner betrieben.

    Die Programme haben jeweils keine Möglichkeit, Daten eines anderen Programms zu sehen oder zu modifizieren

  - **Thread-Parallelität:**

    Innerhalb eines Benutzerprogrammes werden Aktivitäten parallel ausgeführt.

    Die einzelnen Threads innerhalb des Programms greifen auf gemeinsame Daten zu und teilen sich diese.

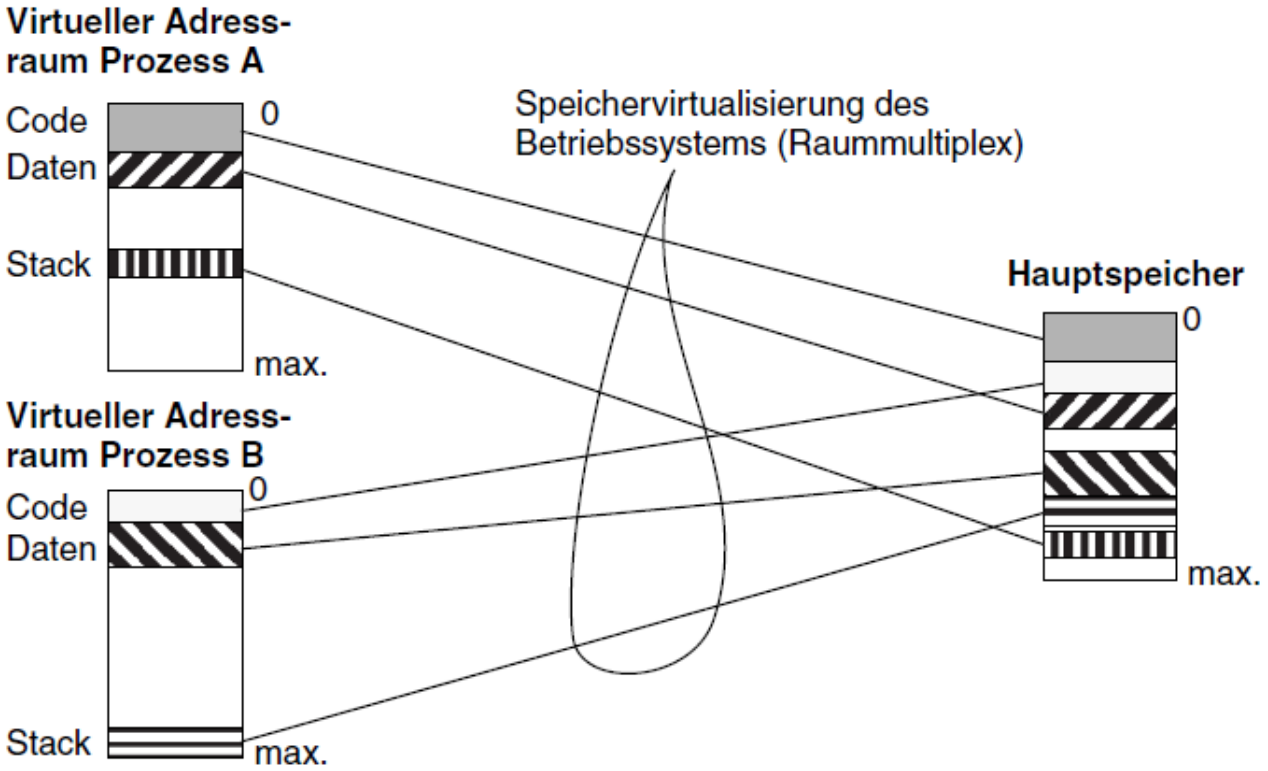
Prozesse vs. Threads (II)

- Aus der Prozess-Parallelität ergeben sich u. a. folgende Anforderungen für die Implementierung des Prozess-Modells:

  - Anwendungs- oder Herstellerübergreifende Nutzung eines gemeinsamen Speicherbereiches gibt es nicht
  - Das Betriebssystem teilt jeder Anwendung (bzw. Prozess) einen eigenen virtuellen Adressraum zu
  - Notwendigkeit der Memory Management Unit (MMU)-Umprogrammierung bei Prozessumschaltung
  - Eine Kommunikation über Anwendungsgrenzen (bzw. Prozessgrenzen) hinweg ist nicht ohne Weiteres möglich (mehr dazu in Interprozesskommunikation)

Prozesse vs. Threads (III)

- Prozess-Parallelität:



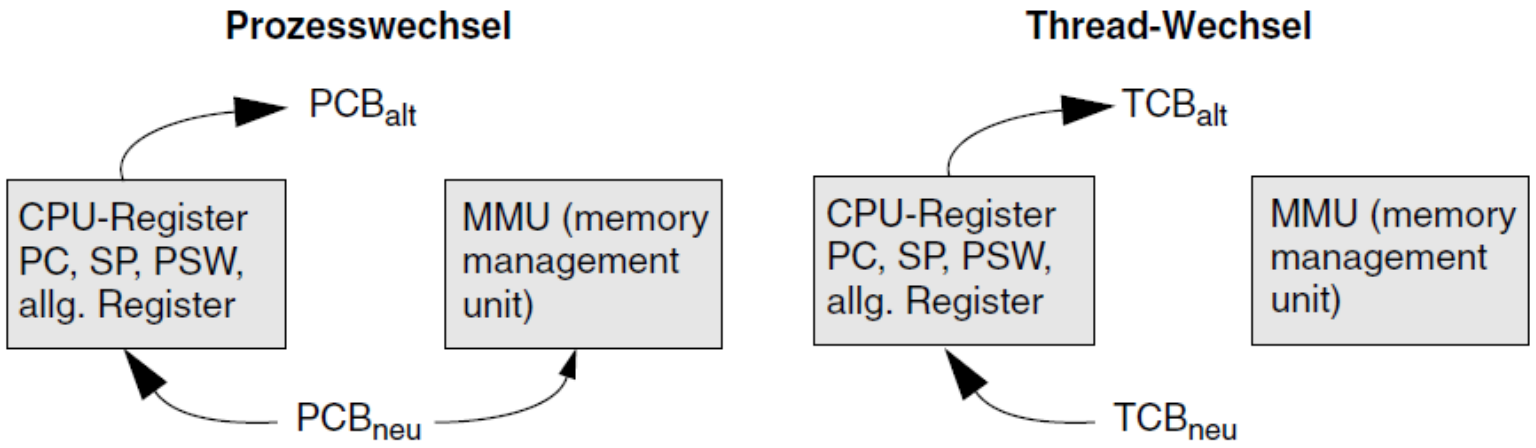
Quelle: *[BS15]*

Prozesse vs. Threads (IV)

- Thread-Parallelität stellt weniger Anforderungen an die Implementierung des Thread-Modells:
  - Keine Notwendigkeit für Bereitstellung eines eigenen virtuellen Adressraums pro Thread
  - Keine Notwendigkeit der MMU-Umprogrammierung beim Thread-Wechsel
  - Einfache Nutzung gemeinsamer Daten über Thread-Grenzen hinweg

Prozesse vs. Threads (V)

- Vergleich: Prozess- bzw. Threadumschaltung



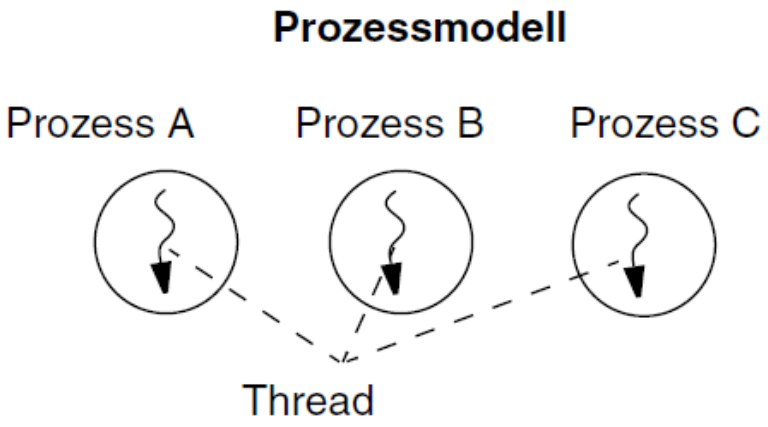
PCB: Process Control Block
(Verwaltungsdaten des Prozesses)

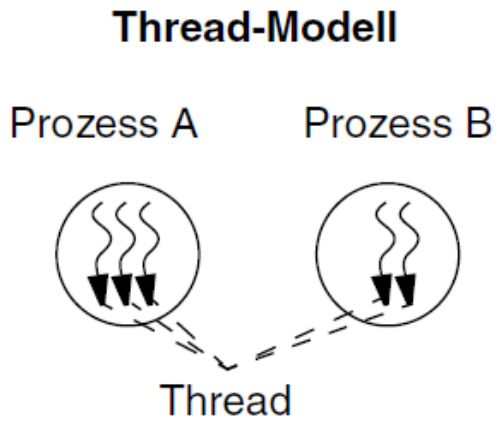
TCB: Thread Control Block
(Verwaltungsdaten des Threads)

Quelle: *[BS15]*

Prozesse vs. Threads (VI)

- Prozess-Modell vs. Thread-Modell





Quelle: *[BS15]*

Implementierung des Multithreading (I)

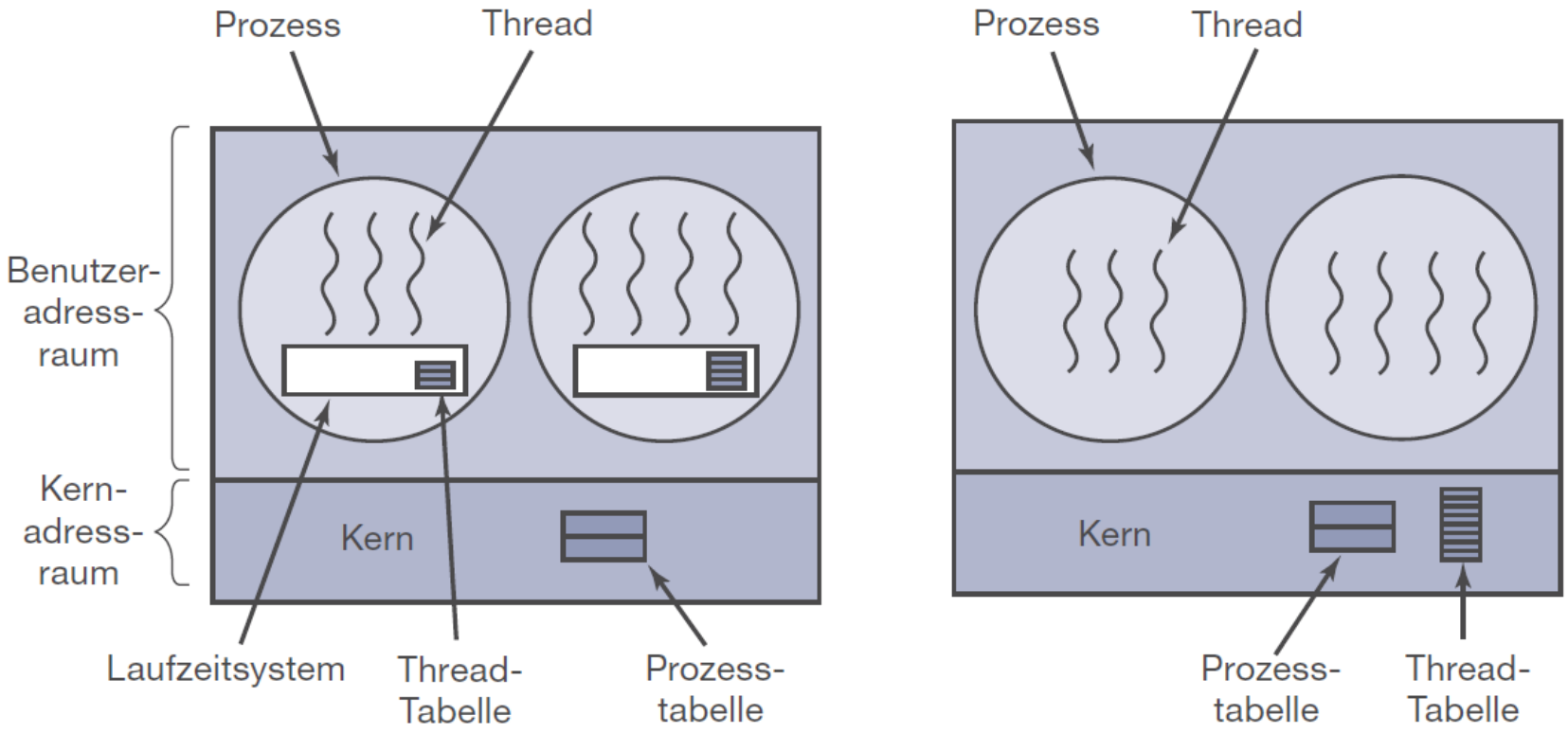
- Implementierung des Multithreading ist systeminterne Angelegenheit, welche vom Betriebssystem abstrahiert wird
- Dabei: In der Anwendungsentwicklung spielt betriebssystemseitige Implementierung hinsichtlich funktionaler Aspekte keine Rolle, da in jedem Fall gewünschte Funktionalität erbracht wird
- Jedoch: Zeitliches Verhalten bei der Umsetzung der Funktionalität hängt von Implementierungsform ab, da hierbei über die Zuteilung der Rechenzeit entschieden wird
- In diesem Zusammenhang ist es wichtig, die unterschiedlichen Implementierungsformen des Multithreading zu kennen, sodass unerwünschte Seiteneffekte ausgeschlossen werden können

Implementierung des Multithreading (II)

- Grundlage: Unterscheidung zwischen zwei Thread-Formen:
  - **User-Level Threads (UL-Threads):**
    Threads, die standardmäßig bei der Anwendungsentwicklung aus Sicht eines Entwicklers verwendet werden. Diese Threads abstrahieren Kernel-Level Threads und werden durch den Programmfluss des Entwicklers selbst verwaltet.
  - **Kernel-Level Threads (KL-Threads):**
    Threads, die tatsächlich auf der CPU ablaufen und vom Betriebssystem verwaltet und gesteuert werden. Diesen Threads teilt das Betriebssystem Rechenzeit zu.
- Daraus folgt: Ein UL-Thread muss auf einen KL-Thread abgebildet werden, bevor er real zur Ausführung gebracht werden kann.

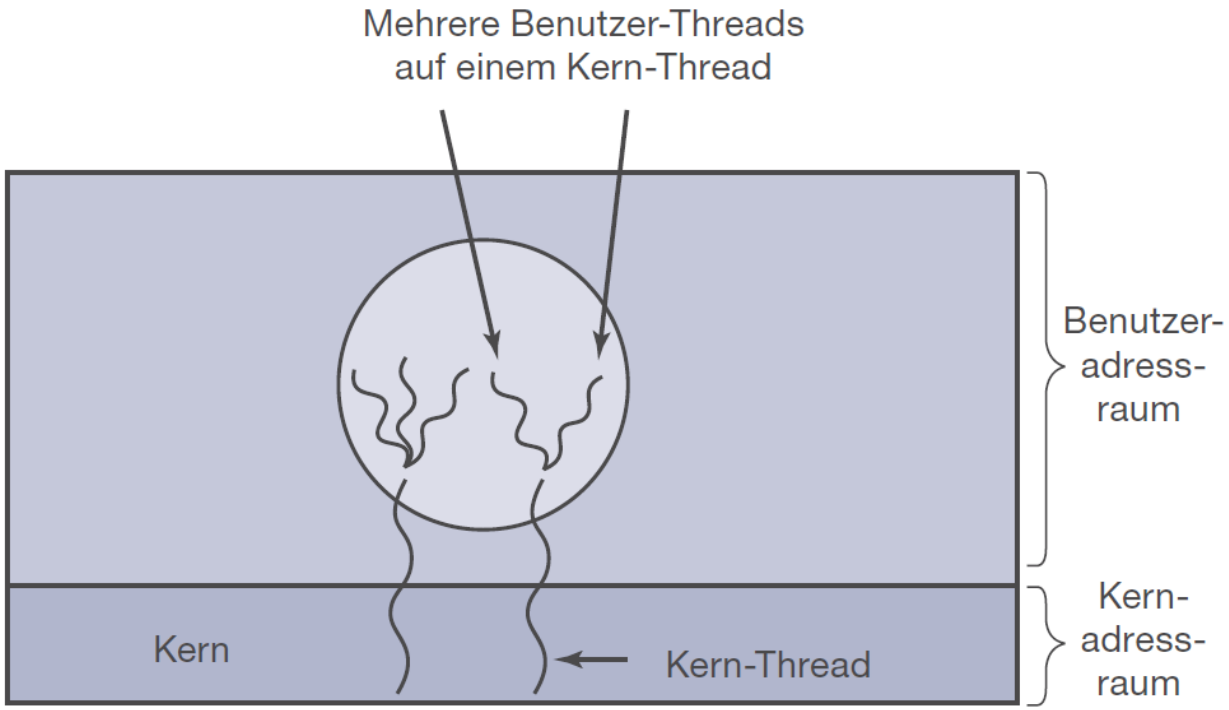
Implementierung des Multithreading (II.1)

- Thread-Verwaltung auf Benutzerebene (links) vs. Thread-Verwaltung auf Kernebene (rechts):



Implementierung des Multithreading (II.2)

- Bündelung von UL-Threads auf KL-Threads:



Implementierung des Multithreading (III)

- Abbildung von User-Level Threads auf Kernel-Level Threads
  - Im einfachsten Fall: Abbildung eines jeden UL-Threads auf jeweils einen KL-Thread
  - Aber: Aus historischen und insbesondere technischen Gründen wird dies nicht immer so umgesetzt
  - Grundsätzlich existieren drei hybride Ansätze zum UL-KL-Mapping:

m:1-Abbildung	**m:n-Abbildung**	**1:1-Abbildung**
Alle UL-Threads eines Prozesses werden auf genau einen KL-Thread abgebildet.	Eine beliebige Anzahl UL-Threads (m) wird auf verschiedene KL-Threads (n) abgebildet. Pro Prozess existieren mehrere KL-Threads.	Jeder UL-Thread wird auf genau einen KL-Thread abgebildet.

  - Im Folgenden: nähere Betrachtung der drei Ansätze

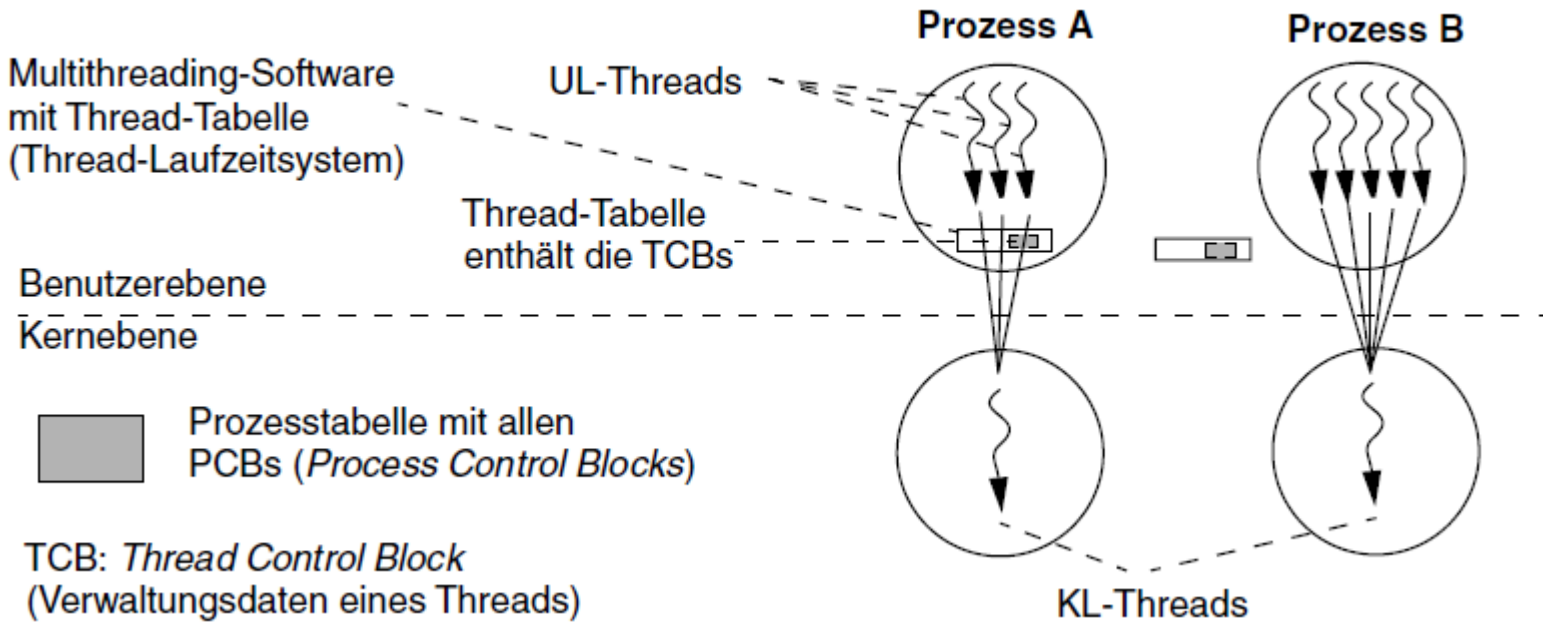
Implementierung des Multithreading (IV)

- **m:1-Abbildung:**

  - **Anwendungsfall:** Betriebssysteme, die kein Multithreading kennen
  - **Funktionsweise:** Betriebssystem teilt den einzelnen Prozessen Rechenzeit zu. Sobald ein Prozess Rechenzeit erhalten hat, wird den einzelnen Threads auf Benutzerebene Rechenzeit zugeteilt.

Implementierung des Multithreading (V)

- **m:1-Abbildung:** Schematische Darstellung:



Quelle: *[BS15]*

Implementierung des Multithreading (VI)

- **m:1-Abbildung:** Vorteile

  - Einzige Lösung für nicht Multithreading-fähige Betriebssysteme
  - Einsatz computersprachbezogener Multithreading-Modelle (Java, .NET) notwendig
  - Sehr schnelle Thread-Umschaltung, da kein Ein- und Austritt im Kernmodus nötig
  - Jeder Prozess realisiert bei Bedarf eigenes Multithreading-Modell
  - Gute Skalierung, da kein Tabellenplatz für Threads im Betriebssystemkern belegt wird

Implementierung des Multithreading (VII)

- **m:1-Abbildung:** Nachteile

  - Bei blockierendem Systemaufruf blockieren alle Threads, da Betriebssystem nur einen einzigen Thread zur Ausführungszeit kennt
  - Bei Seitenfehlern müssen alle Threads innerhalb des betroffenen Prozesses warten
  - Vorteile eines Mehrkernsystems können nicht ausgenutzt werden,
    da alle Threads eines Prozesses zwangsläufig auf einem Kern ablaufen

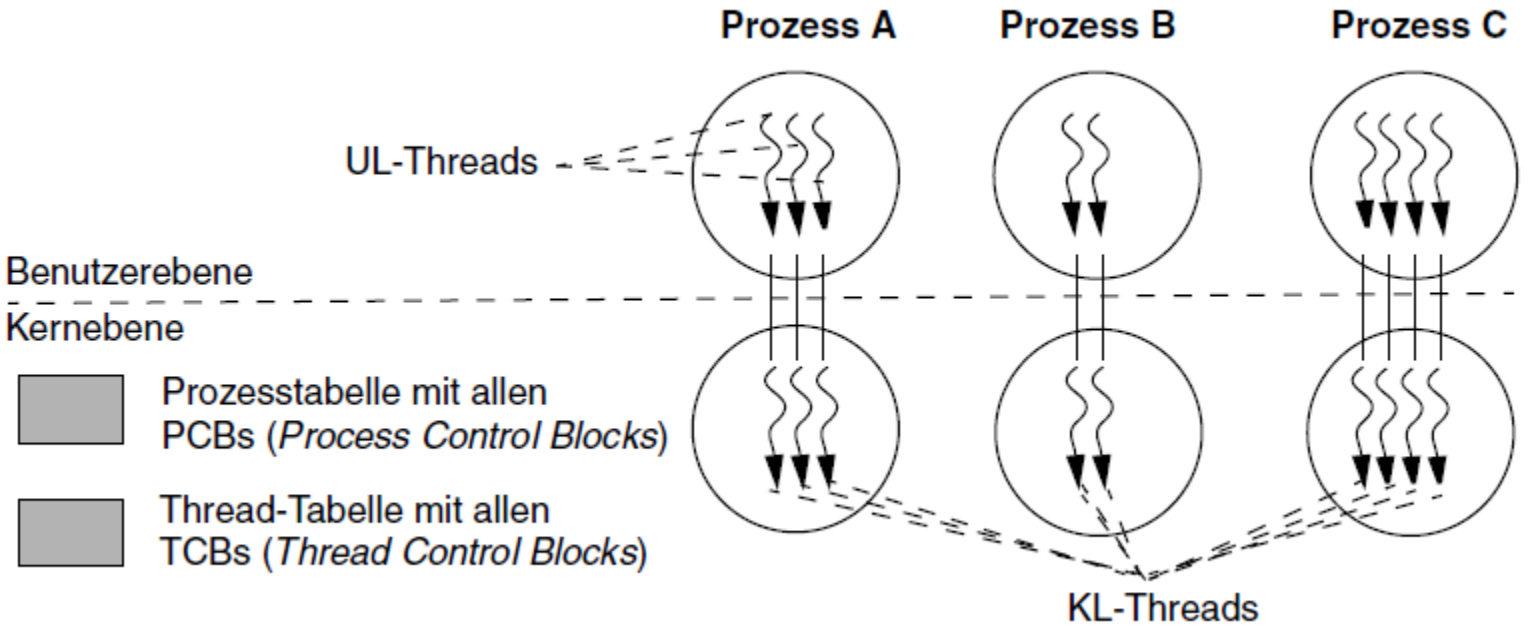
Implementierung des Multithreading (VIII)

- **1:1-Abbildung:**

  - **Anwendungsfall:** Multithreading auf Systemebene
  - **Funktionsweise:** Zuteilung der Rechenzeit findet auf Systemebene und durch das Betriebssystem statt.
    Das Betriebssystem kennt durch die 1:1-Abbildung auch die UL-Threads.

Implementierung des Multithreading (XI)

- **1:1-Abbildung:** Schematische Darstellung



Quelle: *[BS15]*

Implementierung des Multithreading (X)

- **1:1-Abbildung:** Vorteile

  - Nachteile der m:1-Abbildung werden behoben
  - Einheitliche Abbildung für alle laufenden Prozesse
  - Vorteile eines Mehrkernsystems können genutzt werden, da die einzelnen Threads direkt auf CPU-Kerne abgebildet und ausgeführt werden können

Implementierung des Multithreading (XI)

- **1:1-Abbildung:** Nachteile

  - Für Thread-Wechsel ist Umschalten in den Kernmodus notwendig → führt zu Verlangsamung
  - Schlechte Skalierung im Vergleich zur m:1-Abbildung, da für jeden Thread systemintern Ressourcen belegt werden müssen

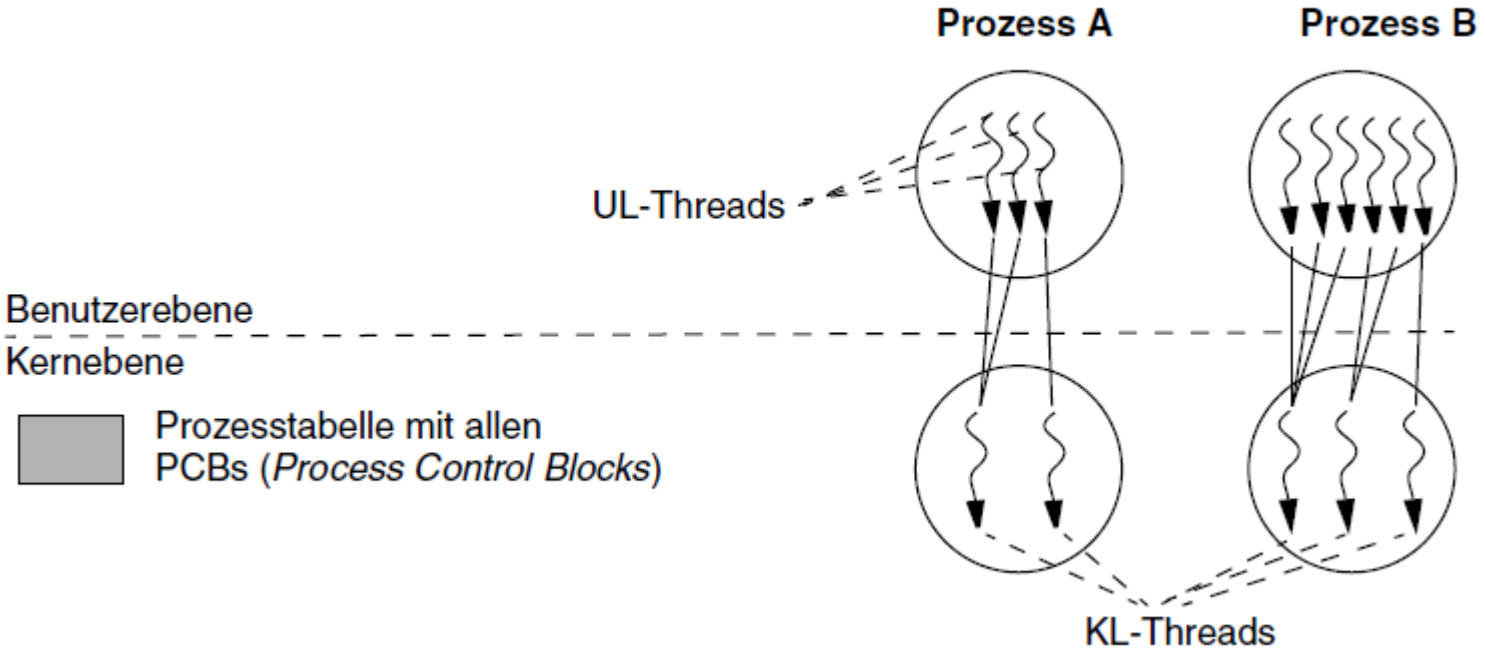
Implementierung des Multithreading (XII)

- **m:n-Abbildung:**

  - **Anwendungsfall:** Mehrkernsystem, auf dem Anwendungen mit mehreren Threads ausgeführt werden
  - **Funktionsweise:** Kombination aus m:1-Abbildung und 1:1-Abbildung

Implementierung des Multithreading (XIII)

- **m:n-Abbildung:** Schematische Darstellung



Quelle: *[BS15]*

Implementierung des Multithreading (XIV)

- **m:n-Abbildung:** Vorteile

  - Entwickler können so viele UL-Threads wie nötig erzeugen, ohne dabei ein maximales Limit zu berücksichtigen (Limitierung ist z. T. bei der 1:1-Abbildung gegeben)
  - Die auf Kernel-Threads abgebildeten UL-Threads können auf einem Mehrkern-System echt-parallel ablaufen
  - Bei blockierenden Systemaufrufen kann das zugrundeliegende Betriebssystem einfach einen anderen KL-Thread zur Ausführung bringen

Quelle: *[BS15]*

Implementierung des Multithreading (XV)

- **m:n-Abbildung:** Nachteile

  - Bei blockierenden Aufrufen unter Verwendung nur eines einzelnen KL-Threads müssen wie bei m:1-Abbildung alle anderen Threads warten → Abhilfe schafft sogenannte `Scheduler Activation`
  - Beim Einsatz auf einem Einkern-System gehen die Vorteile dieses Ansatzes verloren

Quelle: *[BS15]*

Implementierung des Multithreading (XVI)

- Ergänzende Mechanismen für das Multithreading:

1. **Scheduler Activation:**

   Dabei: „Nachahmen“ der Funktionalität und nutzen der Vorteile von Kern-Threads, allerdings mit besserer Performance und größerer Flexibilität von User-Level-Threads.

2. **Pop-up-Threads:**

   Insbesondere in verteilten Systemen von Relevanz, in denen keine blockierenden Threads für das Warten auf eingehende Nachrichten verwendet werden sollen.

   → Anstatt Pull-Semantik Verwendung von Push-Semantik

Implementierung des Multithreading (XVII)

- **Scheduler Activation:** Gründe für den Einsatz:

  (1) Aufwand für die Erzeugung und Zerstörung von Kern-Threads ist hoch, da nicht Prozess-, sondern systemweite Thread-Tabelle geführt und aktuell gehalten werden muss.

  (2) Weiteres Problem (vgl. m:1-Abbildung): Blockiert ein auf einen Kern-Thread abgebildeter UL-Thread, muss ein Laufzeitsystem darüber benachrichtigt werden können, um anderen Thread ablaufen zu lassen

Implementierung des Multithreading (XVIII)

- **Scheduler Activation**
  - Ansatz:
    - Der Kern weist jedem Prozess eine gewisse Anzahl an KL-Threads zu
    - Laufzeitsystem auf User-Level hat diesen KL-Threads die entsprechenden UL-Threads zuzuordnen
    - Auf Einkern-Maschinen: Scheduling (und quasi-Parallelität) der KL-Threads
    - Auf Mehrkern-Maschinen: Echte Parallelität der KL-Threads
    - Stellt der Kern einen blockierenden Systemaufruf durch einen der KL-Threads fest, erfolgt Benachrichtigung an Laufzeitsystem auf User-Level. Dies bezeichnet man als sogenannten `Upcall`.
    - Anschließend kann das Laufzeitsystem den bisherigen Thread als blockiert kennzeichnen und weitere Threads einteilen, ohne dass eine Umschaltung von Benutzermodus in Kernmodus notwendig ist.

Implementierung des Multithreading (XIX)

- **Pop-up-Threads:**
  - Pro Anfrage Erzeugung eines Threads (sogenannter Popup-Thread)
  - Beispielsweise für Dienstanforderungen per Systemmeldung
  - Laufen Pop-up Threads im Kernmodus weisen sie hohes Schädigungspotenzial auf

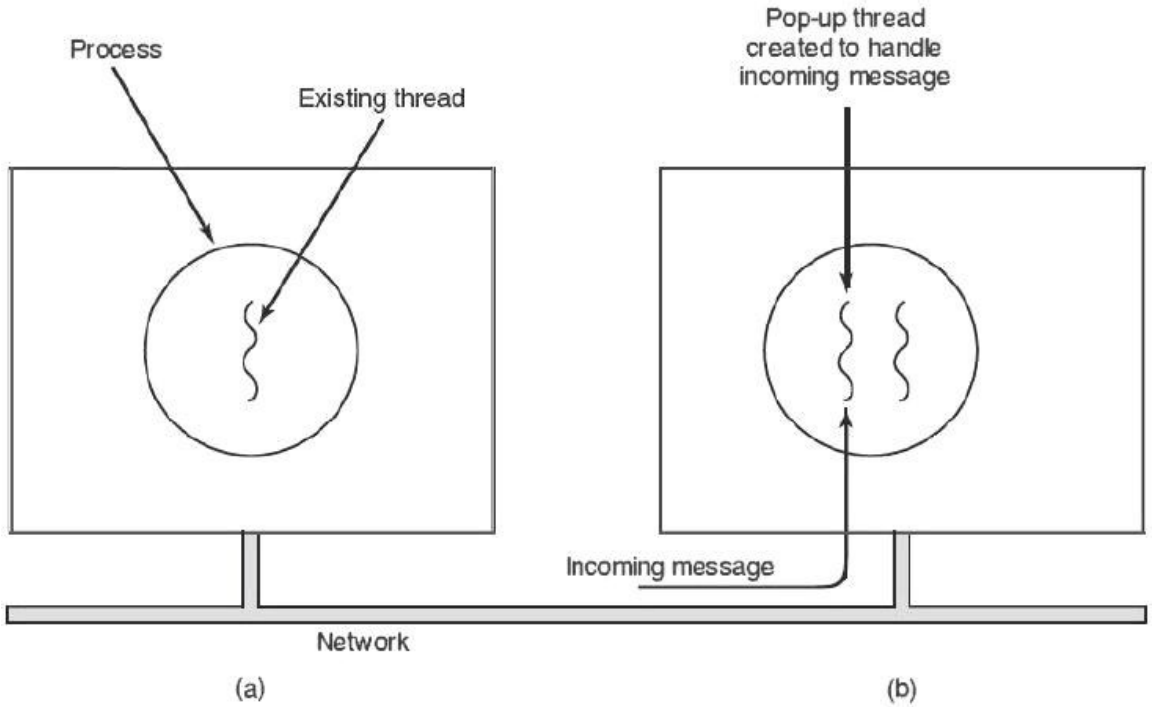


Figure 1: Creation of a new thread when a message arrives.
(a) Before the message arrives. (b) After the message arrives.

Quelle: *Bildquelle: http://www.osinfoblog.com/post/79/pop-up-threads/*

Threads in der Praxis (I)

- Klassische Anwendungsprobleme bei der Verwendung von Threads

1. **Verwendung von Threads zur Parallelisierung von Aufgaben ist gut abzuwägen**
   In diesem Zusammenhang gilt es, auf den Anwendungsfall bezogen ein ausgewogenes Mittelmaß zwischen dem Aufwand zur Aufgaben-Parallelisierung und dem eigentlichen Rechenaufwand zu finden. Dabei gilt:

   ```
   Aufwand zur Aufgaben-Parallelisierung
   + Aufwand für Berechnungen in den einzelnen Threads
   + Aufwand für Zusammenführung der Ergebnisse
   = Gesamtaufwand für thread-parallele Verarbeitung
   ```

   Der Gesamtaufwand für die thread-parallele Verarbeitung sollte wesentlich geringer sein als bei einer single-threaded Berechnung.

Threads in der Praxis (II)

- Klassische Anwendungsprobleme bei der Verwendung von Threads

2. Es können keine Annahmen über zeitliches Verhalten der Verarbeitungsschritte gemacht werden

- Von einem Master-Thread erzeugte Arbeits-Threads müssen nicht zwangsläufig direkt nach der Erzeugung anlaufen.
- Umgekehrt gilt: es muss dennoch berücksichtigt werden, dass ein Thread direkt nach der Erzeugung anlaufen kann, bevor ein Master-Thread weitere Schritte durchführt
- In Multiprocessing-Umgebungen: es können alle Threads echt-parallel ablaufen.

Threads in der Praxis (III)

- Klassische Anwendungsprobleme bei der Verwendung von Threads

3. **Folgerung aus (2): Threads sollten wenn möglich private, isolierte Daten benutzen:**

   Bei der gemeinsamen Datennutzung können keine Annahmen über Ablauf und Zugriffsreihenfolge getroffen werden.

   → Dadurch Gefahr der unsynchronisierten Datennutzung.

   Falls gemeinsame Nutzung von Daten nötig: Verwendung von Synchronisationsmechanismen.

Threads in der Praxis (IV)

- Klassische Anwendungsprobleme bei der Verwendung von Threads

4. **Weitere Folgerung aus (2): Master-Thread sollte Initialisierung von Daten und Objekten bereits vor Anlaufen sämtlicher Threads abgeschlossen haben:**

   - Fehler, der häufig in der Praxis auftritt und z. B. eine NullPointerException nach sich zieht

Threads in der Praxis (V)

- Klassische Anwendungsprobleme bei der Verwendung von Threads

5. Jeder Thread kann jederzeit verdrängt werden

- Dadurch: Keine Zusicherung, dass ein Thread am Stück abläuft
- Arbeitsschritte können unterbrochen werden, wobei sich während der Unterbrechung ggf. gemeinsam genutzte Daten ändern
- Abhilfe: Synchronisationsmechanismen einsetzen

Threads in der Praxis (VI)

- Klassische Anwendungsprobleme bei der Verwendung von Threads

6. Multithreading-Anwendungen laufen je nach Hardware-Gegebenheiten des Systems in zeitlicher Hinsicht unterschiedlich ab

- Dies ist schon gegeben durch quasi-parallele vs. echt-parallele Prozessornutzung
- Effektive Parallelisierung auf dem einen System führt nicht zwangsläufig zu effektiver Verarbeitung auf anderem System

Threads in der Praxis (VII)

- In Java und .NET: Abstraktion von Threads mit Hilfe sogenannter Tasks

  - Tasks bilden Aufgabe ab, nicht Thread-Verwaltung
  - Dadurch: Fokussierung auf eigentliche Entwicklungsaufgaben durch Abstraktion von Threads
  - Programmier- und Laufzeitumgebung sorgt für effizientes Erstellen und Verwalten von Threads, z. B. mit Hilfe von Thread-Pools
  - Tasks bieten erweiterte Möglichkeiten:
    - Task-Ergebnisse
    - Task-Unterbrechung bzw. Abbruch
    - Task-Fortschritt
    - Fortsetzungsaufgaben (Continuation)