

Rechnernetze und Betriebssysteme

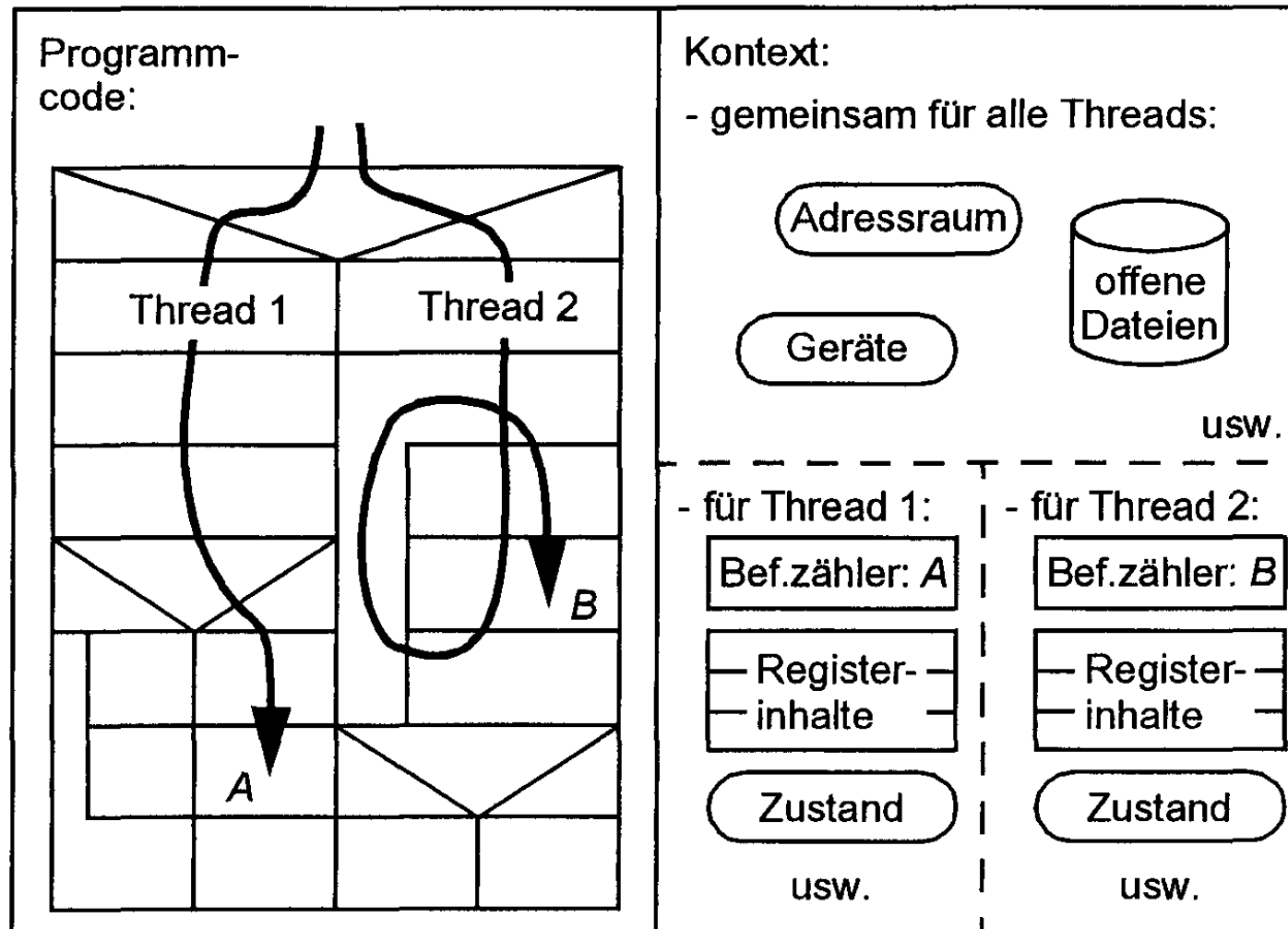
3. Threads

Prof. Dr. Martin Becke
SS2015 – Version 1.0

1. Beschreibung

- Thread (Faden/Strang) ist ein „leichtgewichtiger Prozess“ (lightweight process (LWP))
- 1..n Threads werden immer im Kontext eines Prozesses ausgeführt
- Ein Thread bildet die kleinste Einheit für (OS) Scheduler
- Ein Thread ist kein Prozess, hat aber eine Submenge von eigenen Attribute
 - Eigener Programmzähler
 - Eigener (Thread-)Stack
- Ein Thread teilt sich Eigenschaften mit anderen Threads des gleichen Prozesses
 - Codesegment
 - Gleichen Adressraum (z.B. Globale Variablen vom HEAP)
 - Andere Prozessressourcen

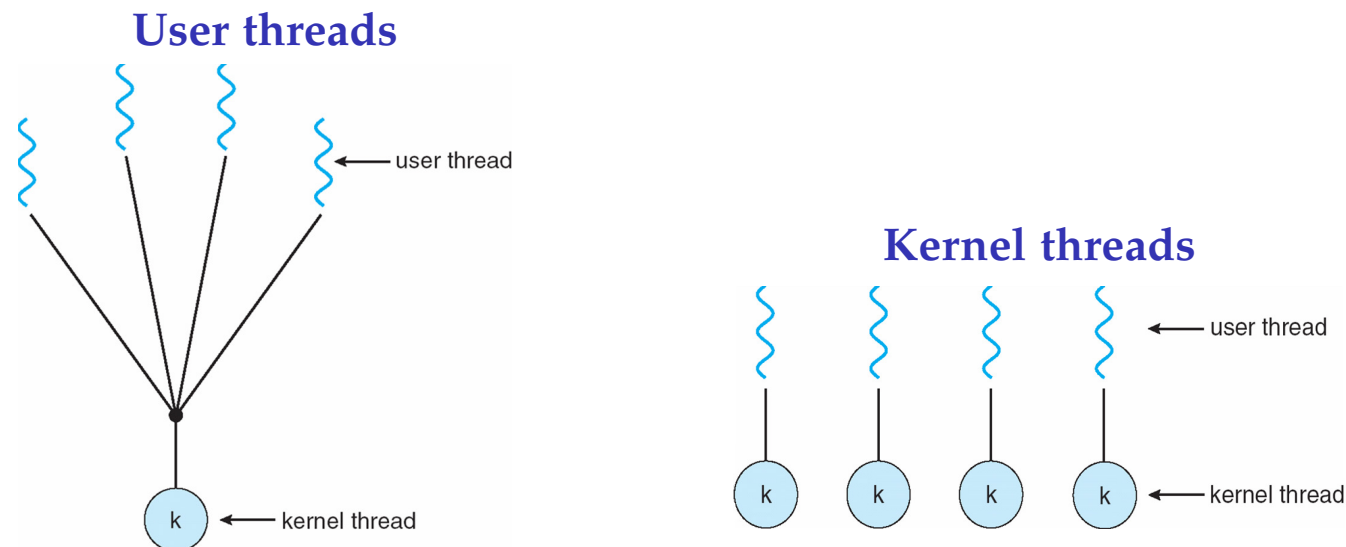
1. Beschreibung




2. Motivation

- Sehr beliebtes Werkzeug um Nebenläufigkeit zu implementieren
 - Echte Nebenläufigkeit mit Kernel Threads
 - Scheduling vom OS (Präemptives Multitasking)
 - Pseudo-Nebenläufigkeit mit User Threads
 - Scheduling von User oder nutzbaren Bibliotheken (Kooperatives Multitasking)
- Erlaubt verschiedenen Code Segmenten Zugriff auf Ressourcen zur gleichen Zeit

3. Kernel vs. User Threads



- Kernel Threads sind Implementiert mittels System Calls
- Kernel Threads sind „leichter“ als Prozesse, aber immer noch mit Kontextwechsel
 - Jeder Thread Aufruf muss durch Kernel
Beispiel: syscall braucht ca. 100 Zyklen, Funktionsaufruf braucht ca. 5
- Kernel Threads können auf verschiedene CPUs/Kerne aufgeteilt werden, weil OS Scheduler sie kennt
- Kernel Threads sind sehr vielseitig einsetzbar, eine Spezialisierung fällt schwer
- User Threads bieten nicht immer (einige Bibliotheken unterstützen es) echte Nebenläufigkeit 
- User Threads benötigen mehr Implementierungsaufwand (eigene Implementierung, Bibliotheken)

4. User Threads Implementierung

- Für jeden Thread muss Speicher vom Wurzelprozess allokiert werden
 - Bei falscher Allokierung ist Stackoverflow nicht unüblich
- Best Practice: Organisiere Thread Pool
- Best Practice: Überschreibe system calls, um blockierende Aufrufe in User Threads zu verlagern (read, write, etc...)
- Optimierte Scheduler nach deiner Bedürfnissen
(Im Hinterkopf behalten: Das OS betreibt weiter präemptives Multitasking)

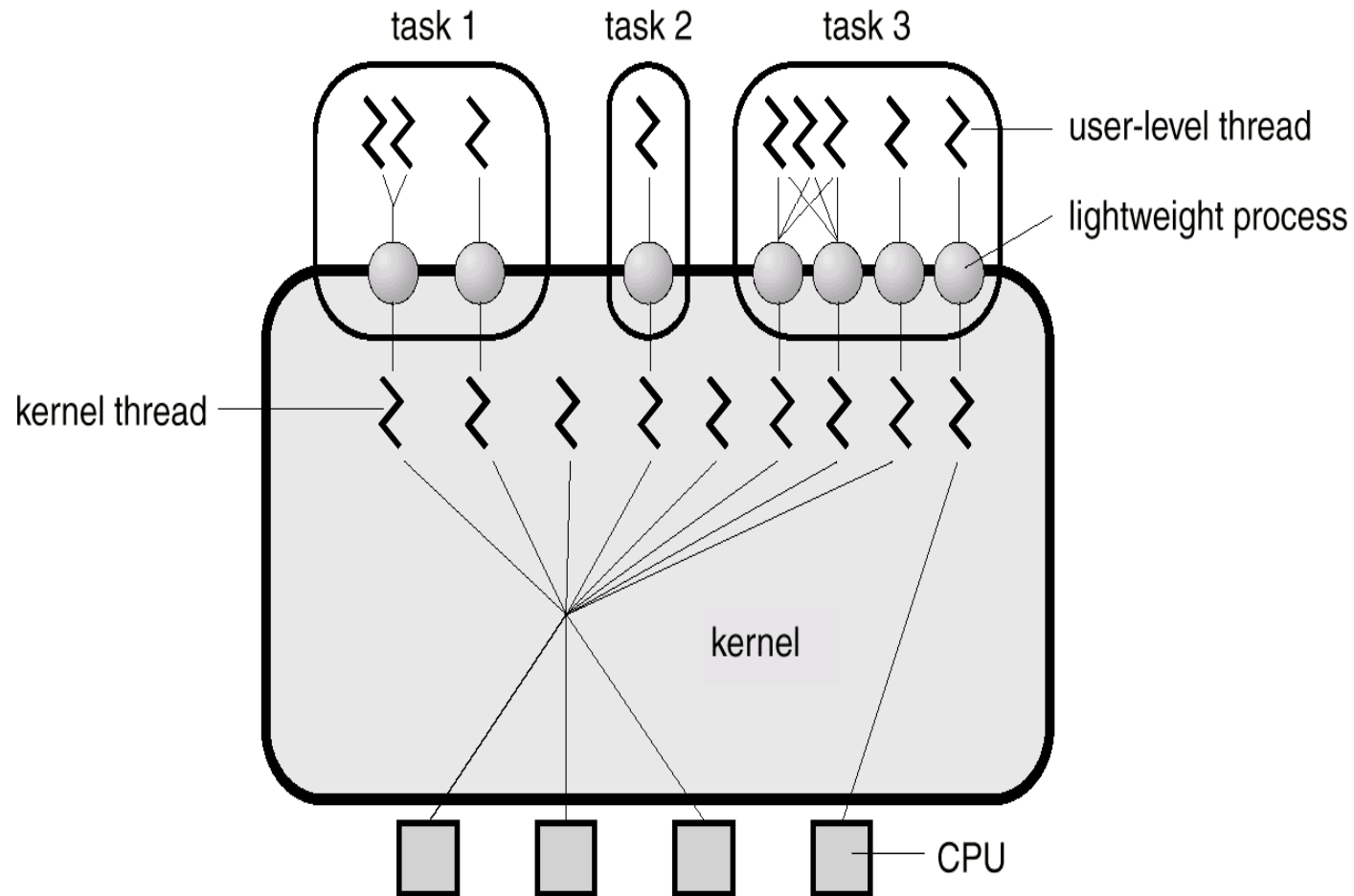
5. Beispiel UNIX/Linux

- Original UNIX kennt keine Threads
- Kernel behandelt Threads wie Prozesse
- Kernel Threads können ähnlich dem Befehl *fork()* mit *int clone(...)* erzeugt werden
- Üblich in Linux ist die Nutzung der POSIX Threads
 - Low Level Thread Bibliothek (IEEE)
 - Nutzt Kernel Level Threads in 1:1 Modell
 - 1:1 Modell → Zu jedem User Level Thread existiert genau ein Kernel Thread
 - Portabel auf POSIX unterstützenden Betriebssystemen
 - Vorsicht mit Stackgrößen
 - C als Sprache und Interface-Beschreibung
 - pthread Header
 - Bibliothek muss beim Compiler angegeben werden (gcc -lpthread)
 - Es fehlen Eigenschaften wie Priorisierung


5. Beispiel UNIX/Solaris

- Solaris nutzt echte LWP
- PCB verwaltet LWP
- LWP kann dauerhaft einer CPU zugeordnet werden
- Solaris organisiert bereitgestellten LWP Pool
- Unterstützt Many-to-Many User/Kernel Thread Modell
 - Ein Prozess kann mehrere Kernel Threads unterstützen, die wiederum mehrere User Level Threads unterstützen

5. Many-To-Many



5. Beispiel Windows

- Windows unterstützt Threadkonzept als Threadobjekte
- Windows bietet eigene (nicht POSIX) API
- Alle Threads sind Kernel Threads
- User Threads können als „Fiber“ Objekte initialisiert werden 
 - Dies entspricht aber einer anderen API

5. Beispiel Java

- Java setzt auf Java Virtual Maschine (JVM) auf
 - JVM ist sehr abhängig vom Host OS
 - Thread Unterstützung nur auf Softwareebene
 - Windows: Java Threads == Kernel Threads
 - Solaris: Java Threads nutzt einen many-to-many LWP Pool
 - Für Nutzung existieren alternativen
 - Ableitung von der „Thread“ Klasse
 - Implementierung des „Runnable“ Interface
 - Implementierung des „Callable“ Interface (Ab Java Version 5)
- ➔ In jedem Fall muss die Methode run() mit Leben gefüllt werden

5. Beispiel Java: Interface vs. Class

- Java unterstützt keine Mehrfachableitungen
 - Thread Ableitung nur sinnvoll, wenn wirklich Methoden überschrieben werden sollen
- Runnable Interface kann auch als „Task“ Implementiert werden.
 - Einfache Interpretation bei Umsetzung von Design Patterns, z.B. Worker Thread
- Task als Runnable implementiert kann mehrmals ausgeführt werden. Ein Thread kann nur einmal gestartet werden.
- Callable ist wie Runnable mit mehr Feedback
 - Mehr Rückgabemöglichkeiten
 - Mehr Fehlerbehandlung

Zusammenfassung

- Beschreibung von Threads
- Motivation von Threads
- Kernel und User Threads
- Beispiele