Thread Programmierung
in Java

Peter Kemper
Lehrstuhl Informatik IV
GB V, R 403, Tel 3031
email: kemper@ls4.cs.uni-dortmund.de
http://ls4-www.cs.uni-dortmund.de/home/kemper/pie.html

Organisatorisches

- Vorlesung: Montags 12-14 Uhr, GB V, HS 113
- Übung: nicht vorgesehen, bei Interesse Projektaufgabe als Gruppenarbeit
- Unterlagen: kein Skript, dafür Folienkopien im Netz unter
  http://ls4-www.cs.uni-dortmund.de/home/kemper/pi.html
  (Vorsicht: Fehler, bitte Literatur nachlesen und Hinweise geben !!!)
- Literatur (in der BI im Semesterhandapparat reserviert)
  - J. Magee, J. Kramer; Concurrency state models & Java programs; Wiley, 1999.
  - S. Oaks, J. Wong; Java Threads, 2nd ed. O´Reilly, 1999.
  - D. Lea; Concurrent Programming in Java, Addison Wesley, 1996.
- Achtung: Sprachumfang Java 1 vs 2 unterschiedlich, einige Methoden deprecated in Java 2, z.B. stop, suspend, resume !

Unterlagen von Magee&Kramer

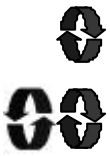
CD-ROM als Buchbeilage, alternativ
http://www-dse.doc.ic.ac.uk/concurrency/

- FSP und LTS Modelle für Beispiele
- Java Beispiele und Demos
- Labelled Transition System Analyser (***LTSA***) zur Modellierung nebenläufiger Prozesse, Animation und funktionaler Analyse von Modellen. Lauffähig unter Windows & Unix, ebenfalls installiert unter /app/unido-i04pub/ltsa/ltsa-v2.0

Danksagung, Credits

- Folien nach Vorlagen von
  - Magee und Kramer
  - Stallings
  - Roy

Was ist ein nebenläufiges/paralleles Programm?



Ein **sequentielles** Programm hat einen sequentiellen Kontrollfluß.

Ein **nebenläufiges / paralleles** Programm hat einen parallelen Kontrollfluß (mehrere sequentielle Krontrollfäden = "Threads of control"), so daß mehrere Rechenschritte simultan durchgeführt und mehrere externe Aktivitäten zur gleichen Zeit kontrolliert werden können.

Schwergewichtige Prozesse vs leichtgewichtige Prozesse (Threads)

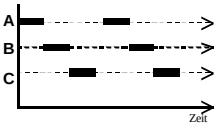
Parallel vs Verteilt

- Paralleles Programm / Parallel Program
  - Programm, in dem an spezifischen Teilen eine parallele Bearbeitung möglich ist
  - Begriff wird typischerweise im Kontext High Performance Computing, Supercomputing, Massiv Paralleler Rechnerarchitekturen verwendet
  - Grundidee: 1 Programm wird von mehreren CPUs (verteilt oder als Multiprozessorarchiktur) simultan abgearbeitet. Dies schließt auch Realisierung als kommunizierende Prozesse (PVM, MPI) ein.
- Verteiltes Programm / Distributed Program
  - Programm, in dem an spezifischen Teilen eine parallele Bearbeitung möglich ist (also wie bei parallelem Programm)
  - Fokus auf verteilter Rechnerarchitektur mit unsicheren Kommunikationsverbindungen, - partnern
  - Resultat der Berechnung wird unter Zuhilfenahme verteilter, unabhängiger Prozesse hergestellt

Concurrent vs Parallel

◆ ***Concurrency / Nebenläufigkeit***
- *Logische Parallelverarbeitung.*

Erfordert nicht mehrere Prozessor einheiten (PEs), aber zumindest verschränkte Abarbeitung auf 1 PE.



◆ ***Parallelität***
- *Reale Parallelverarbeitung.*

Verwendet mehrere PEs und /oder unabhängige, asynchrone Devices.

Nebenläufigkeit UND Parallelität erfordern Zugriffskontrolle für geteilte Ressourcen.
D.h. prinzipielle Probleme gleich, Performance unterschiedlich

Ebenen der Parallelisierung: Granularität

◆ Anwendung
- mehrere Anwendungen laufen parallel ab
- aus Sicht des Betriebssystems: Multitasking, Multiprogramming

◆ Prozeß
- kommunizierende und u.U. konkurrierende Prozesse laufen parallel ab und realisieren 1 Anwendung
- verteilte/parallele Programmierung auf LANs, NOWs und Multiprozessorarchitekturen
- Kommunikation über Sockets oder Message Passing bei Bibliotheken wie PVM, MPI oder geteilten Speicher, remote procedure calls etc

◆ Threads
- innerhalb eines Prozesses laufen u.U. konkurrierende Threads parallel ab, kommunizieren innerhalb eines Adreßraums über geteilten Speicher

◆ Anweisung
- Anweisungen eines Programms werden parallel ausgeführt
- auf dieser Ebene setzen parallelisierende, optimierende Compiler an

◆ Instruktion
- Instruktionen werden parallel ausgeführt, typischerweise parallele Hardware, z.B. floating point Operationen

Threads in JAVA

◆ sind für sich ablauffähige Programmteile, die mit anderen Threads über gemeinsame Objekte kommunizieren

◆ Java Threads werden innerhalb eines Java Programms erzeugt, gestartet, greifen auf Objekte zu, führen Methoden aus, terminieren.

◆ Java Threads sind Bestandteile eines Prozesses (= in Ablauf befindliches Programm) und teilen sich den logischen Adreßraum des Prozesses

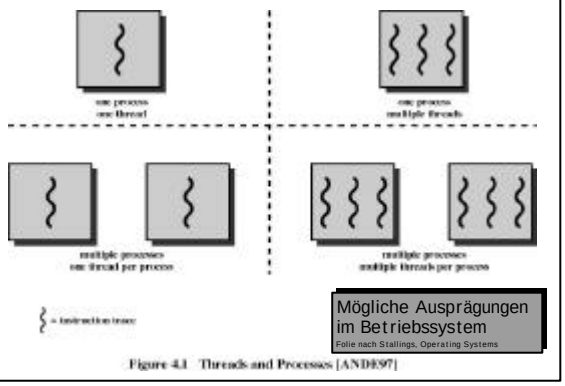
◆ Die reale Hardware, z.B. Anzahl Prozessoren, ist Java Threads unbekannt. Scheduling und Ressourcenzuordnung wird durch die VM oder das Betriebssystem geregelt.

◆ daher: parallele Programmierung mit Threads agiert auf einer virtuellen Shared Memory Architektur

◆ Zur Abgrenzung: in MPI, PVM ist die Sichtweise auf unabhängig agierende Prozesse, die erzeugt werden, einen eigenen, isolierten Adreßraum haben, per Nachrichtenaustausch kommunizieren, ablaufen und terminieren.

Prozesse im Betriebssystem - einige Beobachtungen

- Prozeß durch 2 Eigenschaften ausgezeichnet:
  - Ressourcenbesitz - ein Prozeß erhält einen virtuellen Adreßraum für das Prozeßimage, erhält Zugriff auf Dateien und Geräte über OS
  - Scheduling/Ausführung - führt zu einer Folge von Instruktionen (Trace), über die Zeit mit Traces anderer Prozesse verwoben (Interleaving)
- Beobachtung:
  - Eigenschaften sind unabhängig!!!
  - Daher durch OS unabhängig behandelbar
- Konklusion: Aufteilung resultiert zu Threading
  - Thread bezeichnet Prozeßanteil für das Scheduling/Dispatching
  - Process/Task bezeichnet Prozeßanteil für Ressourcenbesitz



Mögliche Ausprägungen im Betriebssystem
Folie nach Stallings, Operating Systems

Figure 4.1 Threads and Processes [ANDE97]

Multithreading in Betriebssystemen

- OS unterstützt mehrere Threads für die Ausführung eines einzelnen Prozesses, d.h. prozeßinterne Parallelität möglich
- Beispiele:
  - 1 Prozeß, 1 Thread:
    - MS-DOS
  - n Prozesse, je Prozeß 1 Thread
    - UNIX (traditionell)
  - n Prozesse, je Prozeß m Threads
    - Windows 2000
    - Solaris
    - Linux
    - Mach
    - OS/2

Prozeß

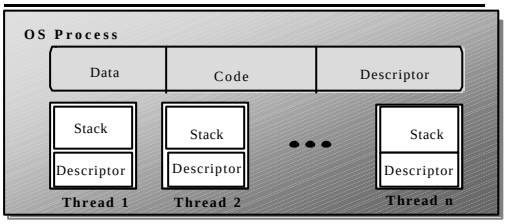
- Virtueller Adreßraum für das Prozeßimage
- Schutz (Protection) beim Zugriff auf CPUs, Interprozeßkommunikation, Dateizugriffe, I/O Ressources

+ darin enthaltene Threads

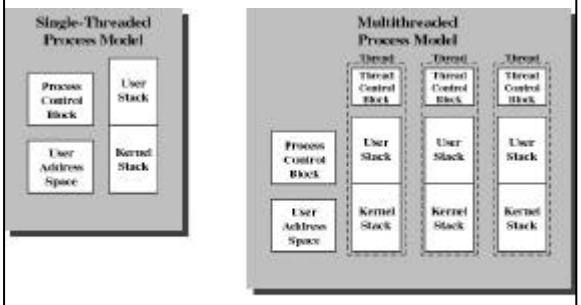
- Ausführungszustand (running, ready, ...)
- Thread Kontext bei Switch gespeichert
- Ausführungsstack
- Statischer Speicherbereich für lokale Variable
- Zugriff auf Speicher und Ressourcen des PROZESSES

  (ACHTUNG: alle Threads teilen Speicher und Ressourcen!!!)

Threads aus Sicht des Betriebssystems



Prozesse werden im Betriebssystem durch Code, Daten und Registerzustand beschrieben. Bei mehreren Threads für einen Prozeß werden dann zusätzlich je Thread ein Aufrufstack und Deskriptor gebraucht.



Folie aus Stallings, Operating Systems

Figure 4.2 Single Threaded and Multithreaded Process Models

User vs Kernel-Level Threads

- User-level Threads
  - Anwendung leistet Thread Management
  - Kernel ist Existenz von Threads unbekannt
  - Beispiel: falls Java Programm mit Threads + Runtime System durch 1 Prozeß realisiert wird
- Kernel-Level Threads
  - Kernel verwaltet Kontext Information für Prozeß und Threads
  - Scheduling erfolgt auf Thread Ebene
  - Beispiele: Windows 2000, Linux, OS/2
- kombinierter Ansatz
  - Thread Erzeugung im Anwendungsbereich
  - Scheduling und Synchronisation mehrheitlich im Anwendungsbereich
  - n-m Abbildung von User-Level Threads auf Kernel-Level Threads
  - Beispiel: Solaris

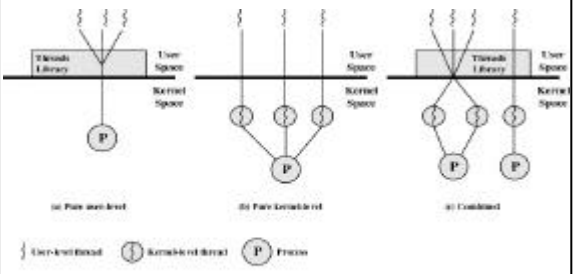


Figure 4.6 User-Level and Kernel-Level Threads

Symmetric Multiprocessing

- Rechnerarchitektur mit n gleichartigen Prozessoren
- Kernel auf jedem Prozessor ausführbar
- Jeder Prozessor führt eigenständig Scheduling durch, wählt Prozesse oder Threads aus einem Pool.
- Unterstützung von Symmetric Multiprocessing ist eine Anforderung an moderne Betriebssysteme
- Symmetric Multiprocessing ist Voraussetzung für hardwareunabhängige parallele Programme.

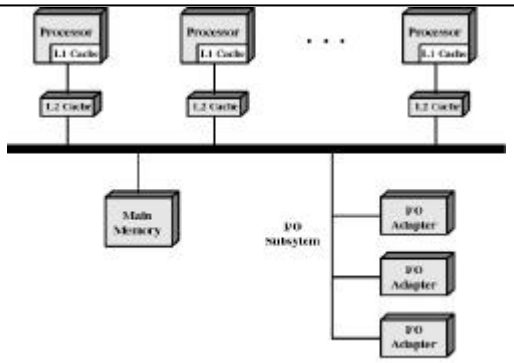


Figure 4.9 Symmetric Multiprocessor Organization

Thread Programmierung: Vorteile

- Kontrollierbarkeit
  - Aktivitäten können von anderen Objekten suspendiert, fortgesetzt oder terminiert werden
- Aktive Objekte
  - Implementierung von Objekten mit unabhängigem, autonomem Verhalten
- Asynchrone Kommunikation möglich
  - Threads können nach Initiierung einer Aktivität fortfahren ohne auf Rückmeldung zu warten
- in Java: Threading ist gekapselt
  - nicht erkennbar, ob Implementierung eines Interfaces intern Threading nutzt oder nicht

Thread Programmierung: Vorteile

- Bessere Performance bei Multiprozessor Systemen
  - Nutzung paralleler Hardware
- höherer Durchsatz einer Anwendung
  - eine I/O Anforderung blockiert lediglich einen Thread
- bessere “Responsiveness” einer Anwendung
  - durch hochpriorigen Thread für Anwender I/O.
  - auch im Sinne von Verfügbarkeit von Diensten
- Programmstruktur problemangemessen
  - für interaktive/reaktive Programme natürlich, Threadstruktur folgt Zerlegung der Teilaufgaben eines Programms (Kontrolle mehrerer Aktivitäten und Ereignisse), z.B. GUI Programmierung.

Thread Programmierung: Grenzen

Threads nicht immer sinnvoll !!!

- Sicherheit/Safety: Koherenz von Daten bei mehreren simultan aktiven Komponenten schwieriger sicherzustellen, Fehlerdetektion schwieriger als bei sequentiellen Programmen, Abhilfe mittels Synchronisationsmechanismen erfordert Sachkenntnis
- Lebendigkeit: Gefahr von Deadlocks (durch Synchronisation) und Starvation
- Nichtdeterminismus
  - Ursache: Interleaving
  - Wirkung: Vorhersage, Verstehen, Debuggen und Reproduzieren von Abläufen schwieriger
- Threads vs Methodenaufrufe
  - Threads nicht bei Request/Reply Strukturen

Thread Programmierung: Grenzen

Threads nicht immer sinnvoll !!!

- Objekte vs Aktivitäten:
  - Anzahl unabhängiger Aktivitäten deutlich geringer als Anzahl Objekte, daher nicht für jedes Objekte Thread erzeugen
- Overhead:
  - für die Thread Erzeugung und Verwaltung, Context-Switching, Synchronization
- Thread vs Prozeß:
  - bei ausreichend großen, unabhängigen Aktivitäten mit eigenen Ressourcen ist Kapselung als eigenständiger Prozeß sinnvoller, Komplexitätsreduktion, Kommunikation über Remote Method Invocation (RMI)

Ist Concurrent Programming praxisrelevant?

Nebenläufigkeit ist verbreitet aber fehleranfällig.

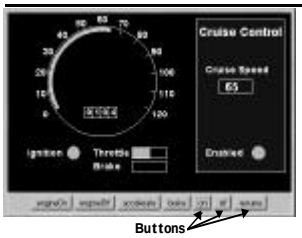
- Therac - 25, eine Gerät zur Bestrahlungstherapie

  Fehler in paralleler Programmierung trugen zu Unfällen mit erheblichen Verletzungen bis zu Todesfolge bei.

- Mars Rover

  Probleme bei der Interaktion zwischen parallelen Tasks führten zu wiederholten Softwareresets und reduzierten Verfügbarkeit des Gerätzes während des Einsatzes.

Beispiel: System zur Geschwindigkeitsregelung

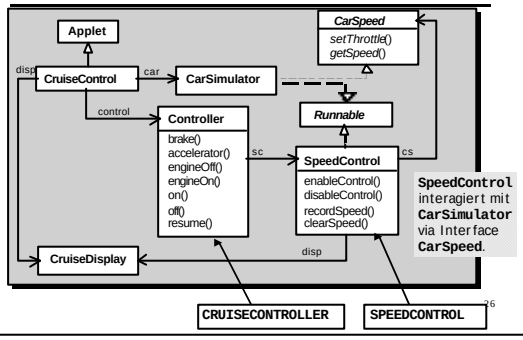


Wenn die Zündung eingeschaltet wird und zusätzlich die Regelung, dann wird die aktuelle Geschwindigkeit gespeichert und die Steuerung hält die Geschwindigkeit ein, bis der Fahrer bremst, beschleunigt oder die Steuerung abstellt.

Mittels **resume** läßt sich die Steuerung reaktivieren.

- ***Ist das System betriebssicher?***
- ***Reicht Testen zur Ermittlung aller Fehler aus?***

Beispiel: Klassendiagramm der Steuerung



SpeedControl interagiert mit **CarSimulator** via Interface **CarSpeed**.

Beispiel: Geschwindigkeitssteuerung

- Anmerkung: Das Programm hat Eigenschaften, die einen Einsatz in einem PKW extrem gefährlich machen, jedoch nur unter speziellen Umständen auftreten!
- Hausaufgabe:
  - Laden Sie das Beispiel zusammen mit dem Quellcode von Magee&Kramers Concurreny Page.
  - Lesen Sie den Quellcode, lokalisieren Sie Codebestandteile, die mit Threads zu tun haben.
  - Versuchen Sie durch Testen Fehler zu ermitteln.
  - Versuchen Sie durch Codereview Fehler zu ermitteln.

Inhaltsübersicht zur Vorlesung

- **Threads in Java, Sprachkonstrukte**
- **Modellierung von Threads mit FSPs**
- **Sicherheit**
- **Lebendigkeit**
- **Zustandsabhängige Aktivitäten**
- **Kontrolle von Nebenläufigkeit**
- **Threads als Anbieter von Diensten**
- **Architekturen auf Basis von Produzenten/Konsumenten**
- **Koordinierte Interaktion**

Entwurf paralleler Programme nach Magee & Kramer

Komplexe Systeme werden in eine Menge einfacherer Aktivitäten zerlegt,die sich als sequentielle Threads notieren lassen. Threads können sich überlappen oder nebenläufig sein, je nach Aufgabenstellung, physischer Rahmenbedingungen, Kommunikationsanforderungen ....

Entwurf paralleler Programme kann komplex & fehlerträchtig sind. Daher systematisches Vorgehen wesentlich.

Konzept: Threads als Folge von Aktionen.

Modell: Threads als Finite State Processes, oder Automaten.

Anwendung: Threads werden Java Threads.

Modellierung

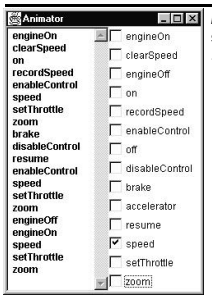
Ein Modell ist eine vereinfachte Repräsentation eines realen Sachverhaltes.

Modellgestütztes Design dient dazu Vertrauen in die Validität eines Entwurfs zu entwickeln.

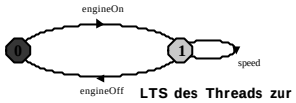
- Fokus auf relevanten Aspekt: hier Concurrency
- Modellanimation zur Visualisierung des Verhaltens
- Automatische Verifkation von Eigenschaften (Sicherheit & Fortschritt)

Modelle werden durch Automaten beschrieben (Labelled Transition Systems **LTS)**. Diese werden textuell als Finite State Processes (**FSP**) notiert und durch das ***LTSA*** Werkzeug angezeigt.

Modellierung der Geschwindigkeitssteuerung



***LTSA* Animator verfolgt schrittweise Aktionen und Zustände.**



LTS des Threads zur Geschwindigkeitsüberwachung.

Im Rahmen der Vorlesung wird FSP und LTSA in einigen Fällen zur Modellierung, Animation und Verifikation von Entwürfen eingesetzt.

Entwurf paralleler Programme nach D. Lea

- Aufbau einer Methodik
  - zur Unterscheidung von Anforderungen
    - z.B. Sicherheit, Lebendigkeit etc.
  - zur anforderungsabhängigen Angabe von Designmustern, - strategien, „Patterns“, die in einem Entwurf helfen Anforderungen zu erfüllen
- Beide Ansätze ergänzen sich:
  - Patterns hilfreich für Designentscheidungen, zur Erstellung eines Entwurfes
  - modellgestütztes Vorgehen zur genaueren Betrachtung, Bewertung eines Entwurfes

Praktische Einsatzmöglichkeit- Java

Java ist

- erhältlich, akzeptiert, portable
- enthält ausreichend Unterstützung für Concurrent Programming
- aus dem Grundstudium bzgl OO Entwurfstechniken bekannt

Akademische Spielbeispiele erlauben

Konzentration auf Ursachen für Schwierigkeiten bei Nebenläufigkeit.

Lernziel

Diese Vorlesung soll ein Grundverständnis von den Konzepten, Modellen und der Realisierung von parallelen Programmen mit Threads in Java vermitteln.

Die Betonung von Prinzipien, Konzepten und Strategien vermittelt Kenntnisse der Schwierigkeiten / Herausforderungen und der Lösungstechniken. Modellierung bietet einen Einblick in das Verhalten nebenläufiger Systeme und hilft ein Design zu validieren. Programmierung in **Java** bietet dann die Programmierpraxis und das Gewinnen praktischer Erfahrungen in der Umsetzung.

Exkursion in Betriebssysteme (OS)

- Microkernel Architekturen haben Thread Management, Thread Scheduling
- Grundkonzept Microkernel:
  - Kleiner OS Kern aus wenigen essentiellen OS Funktionen
  - viele traditionelle OS Funktionen werden externe Subsysteme, z.B.
    - Gerätetreiber (device driver)
    - Dateisysteme (file systems)
    - Virtual Memory Manager
    - Windowing System
    - Security Services

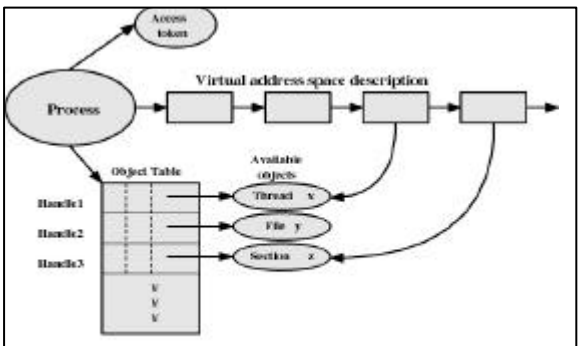
Vorteile des Microkernels

- Uniforme Schnittstelle für Anforderungen durch einen Prozeß (Message Passing)
- Erweiterbarkeit und Flexibilität
  - erlaubt Hinzufügen weiterer Dienste
  - Entfernen von Diensten
- Portabilität
  - Anpassung lediglich im Microkernel
- Zuverlässigkeit
  - modulares Design
  - bessere Testmöglichkeiten bei kleinem Kernel
- Verteilte Systeme
  - Message Passing Konzept für lokale oder verteilte Funktionen gleich.
- Objekt-orientiertes OS
  - Komponenten sind Objekte mit klar definierten Schnittstellen, Software durch Verknüpfung wohldefinierter Teile herstellbar

Microkernel Aufgaben

- Low-level Memory Management
  - Abbildung von Seiten des virtuellen Adreßraums in physikalische Seiten (Page frames)
- Inter-Prozeß Kommunikation
- I/O und Interrupt Management

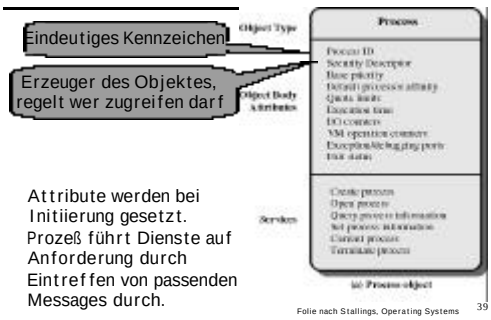
- Beispiel für (modifizierte) Microkernel Architektur:
  - Windows 2000



Folie nach Stallings, Operating Systems

Figure 4.12 Windows 2000 Process and Its Resources

Windows 2000: Prozess Objekte



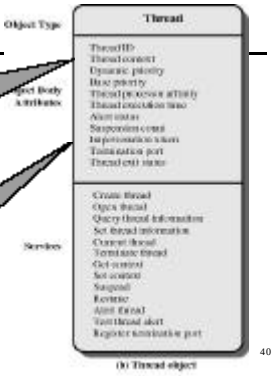
Attribute werden bei Initiierung gesetzt.
Prozeß führt Dienste auf Anforderung durch Eintreffen von passenden Messages durch.

Folie nach Stallings, Operating Systems

Windows 2000
Thread Objekte

Kontextinformation: Registerinhalte etc um Thread Switching zu ermöglichen

Temporäres, geliehenes Zugriffsrecht für die Nutzung anderer Prozesse



(b) Thread object

Windows 2000: Thread Zustände

- Ready: Thread könnte ausgeführt werden, Dispatcher wählt gemäß Priorität aus Threads dieser Klasse aus
- Standby: Für eine CPU bereits selektierter Thread, der auf Freigabe der CPU wartet, bei ausreichender Priorität Preemption!
- Running: Thread wird aktuell ausgeführt
- Waiting: Thread wartet auf I/O, auf Synchronisation oder aufgrund einer Anweisung eines Subsystems.
- Transition: Thread verläßt Waiting Status, aber Ressourcen nicht verfügbar.
- Terminated: freiwillig oder wg Terminierung des Prozesses

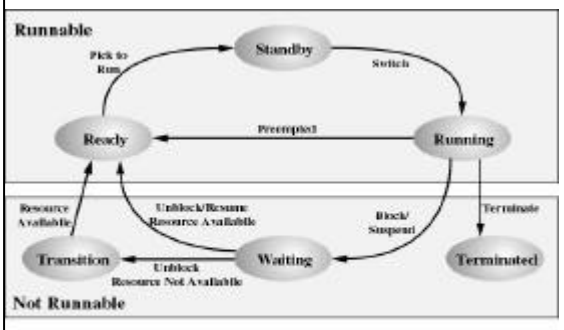


Figure 4.14 Windows 2000 Thread States

Beispiel für andere OS Architekturen: Solaris

- Vier unterschiedliche Konzepte
  - Prozeß: normaler UNIX - Prozeß, enthält den Adreßraum der Anwendung, Stack und Prozeßkontrollblock
  - User-level Threads aufgrund einer Thread Bibliothek innerhalb eines Prozesses, für OS unsichtbar
  - Leichtgewichtige Prozesse als Abbildung von User-Level Threads (n:1) auf Kernel Threads (1:1). Diese Prozesse unterliegen Scheduling des Kernels.
  - Kernel Threads: unterliegt Scheduling/Dispatching auf einer CPU.

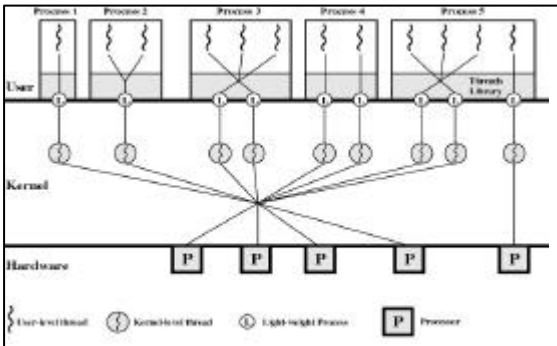


Figure 4.15 Solaris Multithreaded Architecture Example

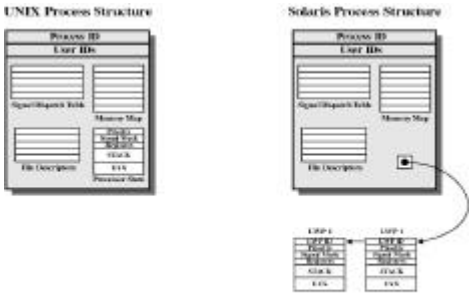


Figure 4.16 Process Structure in Traditional UNIX and Solaris [LEWI96]

Solaris: Ausführung eines User-Level Threads

- Sichtweise: Leichtgewichtiger Prozeß (LWP) des OS im Zustand Running wird unter den ablauffähigen Threads herumgereicht.
- Situationen zum Weiterreichen des LWPs:
- Synchronisation = Warten auf Eintreten einer Bedingung:
  - Thread wechselt zu Sleeping, bei Eintreten der Bedingung zu Runnable;
- Suspendierung = durch Einwirken eines Threads ausgelöst
  - Thread wechselt zu Stopped bis Continue Anforderung eingeht, dann Wechsel zu Runnable
- Unterbrechung (Preemption) = bei Aktivierung eines höherpriorigen Threads
  - Thread wechselt zu Runnable
- freiwillige Kontrollweitergabe (Yielding) = selbstverursacht (thr_yield())
  - Thread wechselt zu runnable, sofern vorhanden erhält anderer Thread (gleicher Priorität) LWP

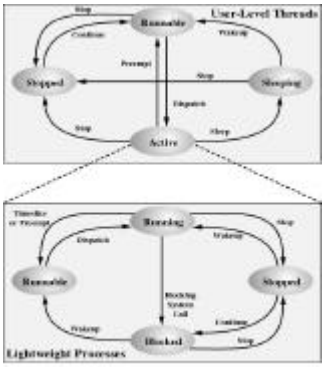


Figure 4.17 Solaris User-Level Thread and LWP States

Linux Prozesse

- Zustand
- Scheduling Information
- Identifier
- Interprozeßkommunikation
- Links/Pointer zum Parent, Sibblings (Geschwister) und Child Prozessen
- Zeiten und Timer
- File System, Zeiger auf geöffnete Dateien
- Virtual memory
- Prozessorspezifische Kontextinformation

- Keine Unterscheidung von Threads und Prozessen
- geteilter Speicher, geteilte Ressourcen entstehen durch Klonen von Prozessen

Linux Prozeßzustände

- Running: zwei Teilzustände für ausführbare Prozesse
  - ready
  - executing
- Interruptable: Zustand, in dem blockierter Prozeß auf ein externes Ereignis wartet, z.B. I/O Ende, Signal eines anderen Prozesse, Ressourcenzuteilung
- Uninterruptable: Zustand, in dem blockierter Prozeß ausschließlich auf Eintreten einer Hardware Bedingung wartet.
- Stopped: Prozeß blockiert und kann nur durch Aktionen von anderen Prozessen fortgesetzt werden, z.B. beim Debugging
- Zombie: terminierter Prozeß, für den noch Datenstrukturen aufrechterhalten werden.

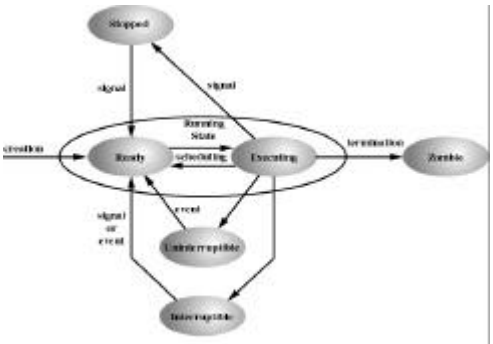


Figure 4.18 Linux Process/Thread Model

Vorteile von Threads

- Behandlung von Threads weniger zeitaufwändig
  - bei Erzeugung
  - bei Terminierung
  - bei Switch (sofern innerhalb eines Prozesses)
- Wegen geteiltem Speicher und geteilter Ressourcen ist Interthreadkommunikation ohne Kernelbeteiligung möglich

Threads im Single-User Multiprocessing System

- Aufteilung der Aufgaben in Vordergrund- (Interaktiv) und Hintergrundaufgaben (Berechnungsintensiv)
- Ausnutzung von Hardwareparallelität
- Asynchrone Aufgaben isoliert programmierbar und ausführbar
- Modulare, problemadäquate Programmstruktur möglich

- Scheduling und Dispatching erfolgt für Threads jeweils einzeln
- Suspendieren des Prozesses impliziert Suspendieren aller Threads wg Sharing
- Terminierung des Prozesses impliziert Terminierung aller Threads