Betriebssysteme Vorlesung 3

Prozeßmanagement II

Threads und Threadmanagement

Prozessmanagement auf Multiprozessorarchitekturen

Literatur: Stallings Kapitel 4

Übersicht

- Konzepte zur Beschreibung von parallelen Anwendungen
  - Prozesse
  - Threads
- Threads:
  - Was ist das ?
  - Wofür setzt man Threads ein ?
  - Wie programmiert man mit Threads in Java ?
- Betriebssysteme und Threads
  - Wie werden Threads durch Betriebssysteme unterstützt ?
  - Wie wird eine Mehrprozessorarchitektur für Threads genutzt ?
    - Symmetrischer Mehrprozessorbetrieb
  - Wie werden Prozesse/Threads zur Gestaltung von Betriebssystemen genutzt ?
    - Microkernel Architektur
- Threadmanagement in realen Betriebssystemen
  - Windows 2000

Nutzen

- Mit Threads lernen Sie ein weiteres Konzept zur Beschreibung paralleler Abläufe kennen.
- Sie lernen elementare Schritte zur Programmierung mit Threads in Java.
- Sie erfahren, wie Threads in unterschiedlichen Betriebssystemen und auf unterschiedliche Art und Weise unterstützt werden.

Der PCB = Process Control Block beinhaltet Verwaltungsinformation für das OS:

1) Identifizierung, (wer ist das?)
2) CPU Status (wo weitermachen?)
3) Kontrollinformation (wann auswählen?)

Aufrufstack der Funktionen/Methoden im aktuellen Zustand der Bearbeitung

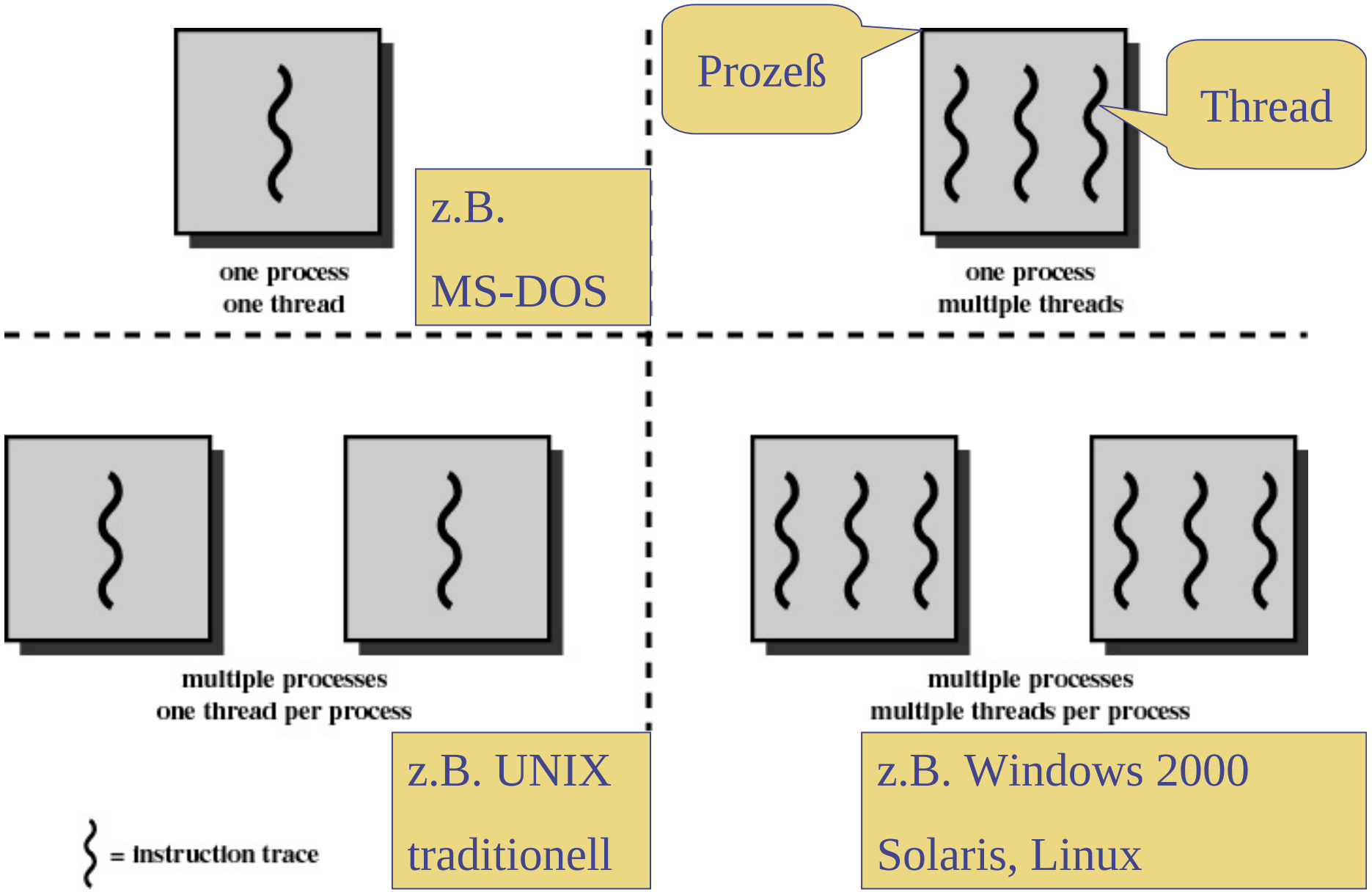
Speicherbereich für Daten/Objekte, Code

Prozeßimage bildet logischen Block, der in der Realität aus einzelnen Speicherseiten (Pages) zusammengesetzt sein kann.
Je nach OS kann noch zusätzlich ein Kernelstack im Prozeßimage auftreten.
Prozeßmanagement arbeitet mit Prozeßtabellen und den PCBs der einzelnen Prozesse.

Prozeß: Ressourcen + Instruktionsfolge

- Ressourcenzuordnung und Ausführung der einzelnen Operationen sind durch OS unabhängig von einander behandelbar
- Trennung führt zum Konzept des “Threads”
  - Thread = Kontrollfaden
  - Threads sind parallel bearbeitbare Anteile eines Programms
  - Threads sind selbst sequentiell, werden zur Laufzeit erzeugt, initialisiert, abgearbeitet, terminiert.
  - Threads kommunizieren über geteilte Speicherbereiche.
- Warum braucht man ein weiteres Konzept ? Mit Prozessen und Interprozesskommunikation lassen sich doch parallele Programme bereits implementieren ?

  Threads eines Prozesses agieren im selben Adressraum, kommunzieren über geteilten Speicher
  -> Threadwechsel für OS einfacher / schneller
  -> reduziertes Sicherheitskonzept zwischen Threads
  -> gut in bestehendes Sprachkonzept integrierbar (JAVA)



Multithreading in OS

OS Feature: Multithreading

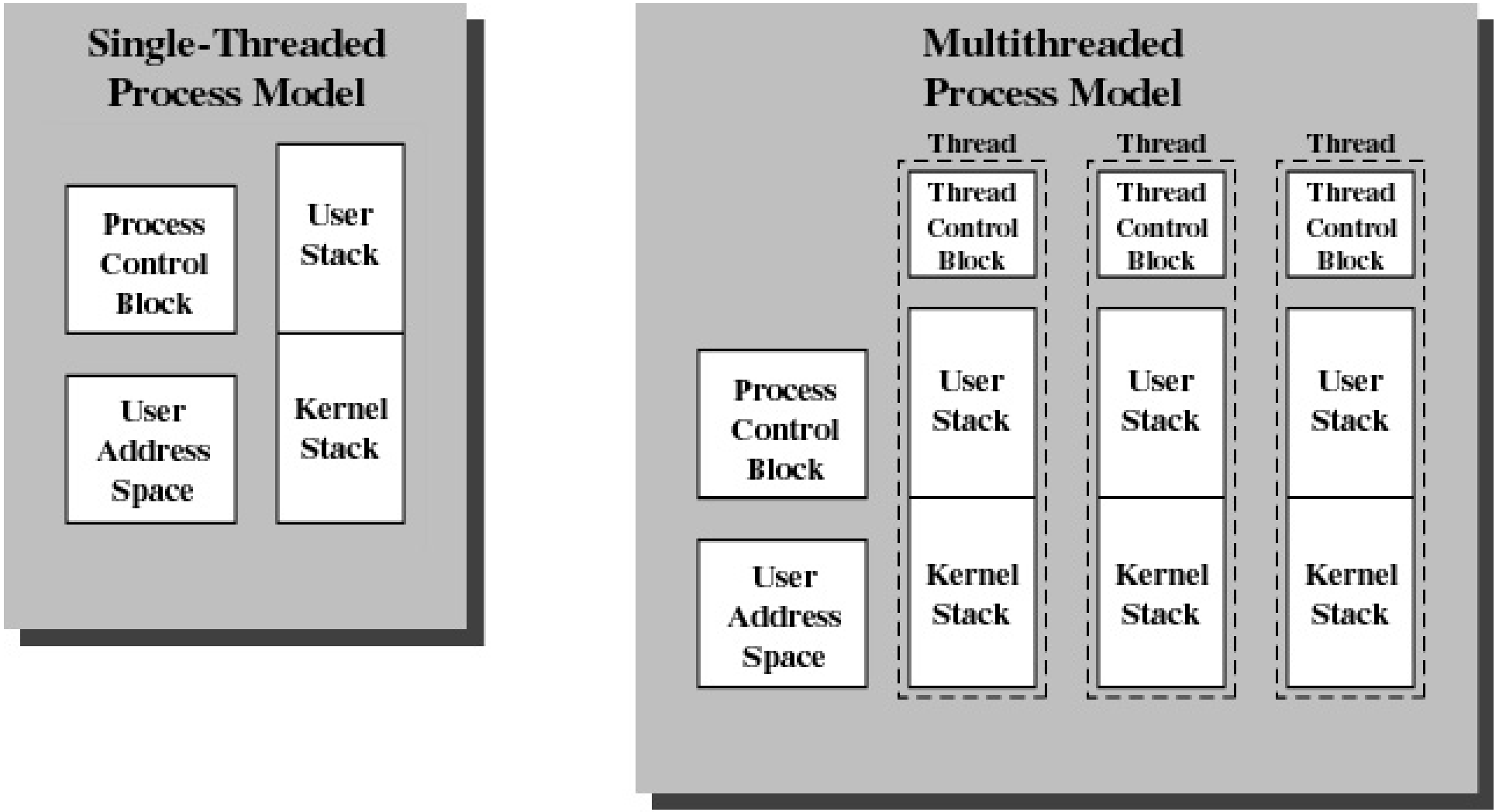
In einem OS mit Multithreading wird die bisherige Prozeßcharakterisierung aufgeteilt:

Ein Prozeß erhält:

- einen virtuellen Adreßraum mit Prozeßimage
- geschützten Zugriff auf CPUs, Dateien, E/A Geräte und andere Prozesse (wg Interprozeßkommunikation)

Innerhalb des Prozesses existieren N≥1 Threads, die jeweils

- einen Threadausführungszustand haben
- einen Kontext haben, der gespeichert wird, wenn der Thread nicht bearbeitet ist (vglbar zum PSW im PCB)
- einen Ausführungsstack
- eigenen, statischen Speicher für Threadlokale Variable
- Zugriff auf den Speicherbereich und alle Ressourcen des Prozesses haben (gemeinsam mit allen Threads des Prozesses)



Stallings Fig. 4.2 Prozeßmodell ohne Threads und mit Threads

je Prozeß: Verwaltung (PCB) + Ressourcen/Adreßraum

je Thread: Verwaltung (TCB) + Thread-interner Aufrufstack

Welche Operationen führt ein OS mit Threads durch ?

- Spawn
  - bei der Erzeugung eines Prozesses wird typischerweise ein erster Thread miterzeugt. Zur Laufzeit kann dieser Thread dynamisch weitere Threads erzeugen. Diese erhalten einen Instruktionszeiger und Argumente/Parameterwerte. Das OS reserviert Speicherplatz für den Registerkontext und Stack des Threads und plaziert den Thread in der Ready-Schlange.
- Block
  - wenn ein Thread anhand eines Befehls (genau wie ein Prozess vorher) warten muss, blockiert er. Registerkontext, Programmzähler und Stackpointer werden gesichert. Das OS entscheidet über die Bearbeitung eines anderen Threads dieses Prozesses aus der Ready-Schlange oder eines anderen Prozesses.
- Unblock
  - bei Eintreten des erwarteten Ereignisses wird ein Thread von der Blocked-Schlange in die Ready-Queue verschoben
- Finish
  - bei Terminierung eines Threads wird der Speicherplatz für Registerkontext und Stack freigegeben

Bewertung des Threadkonzeptes

Pro:

- Performancegewinne bei der
  - der Erzeugung eines Threads,
    - es wird kein neues Prozßimage angelegt
  - der Terminierung eines Threads,
    - es wird nur beim letzten Thread ein Prozeßimage abgebaut
  - dem Wechsel von Threads innerhalb eines Prozesses,
    - es wird im wesentlichen der CPU Kontext gesichert, weniger Datenstrukturen aktualisiert
  - der Kommunikation zwischen Threads innerhalb eines Prozesses
    - es wird nicht jeweils der Kernel bemüht, um Sicherheitskonzepte zu prüfen und die Kommunikation herzustellen, sondern lediglich auf geteilten Speicher zugegriffen

  im Vergleich zu gleichwertigen Operationen für Prozesse
- ermöglicht einfachere Programmstrukturen, Softwarearchitekturen bei nebenläufigen Aktivitäten in Programmen

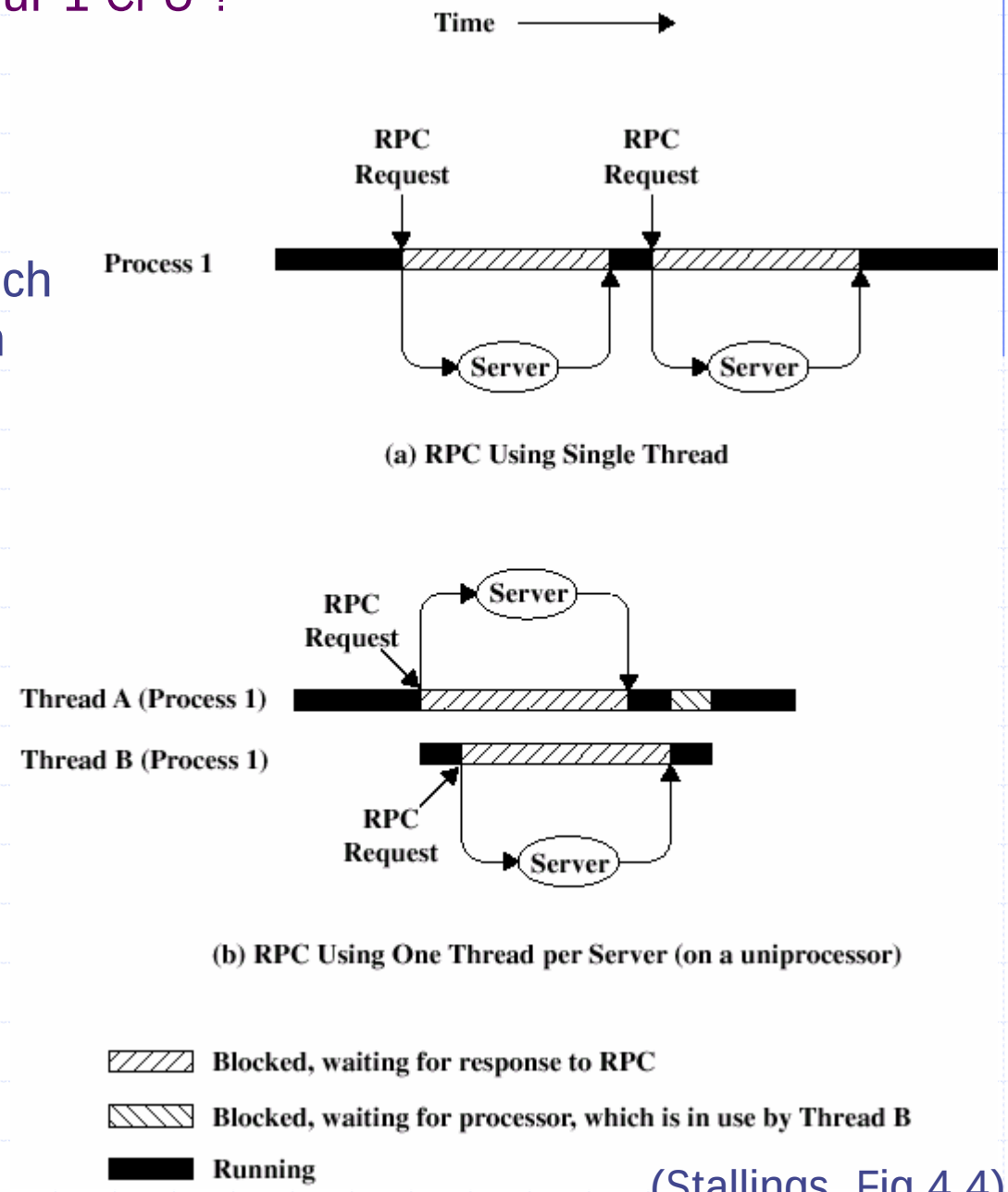
Contra:

- erweiterte Verwaltungsdatenstrukturen für Prozesse
- Threadorientiertes Schedulingverfahren

Threads sinnvoll auch bei nur 1 CPU ?

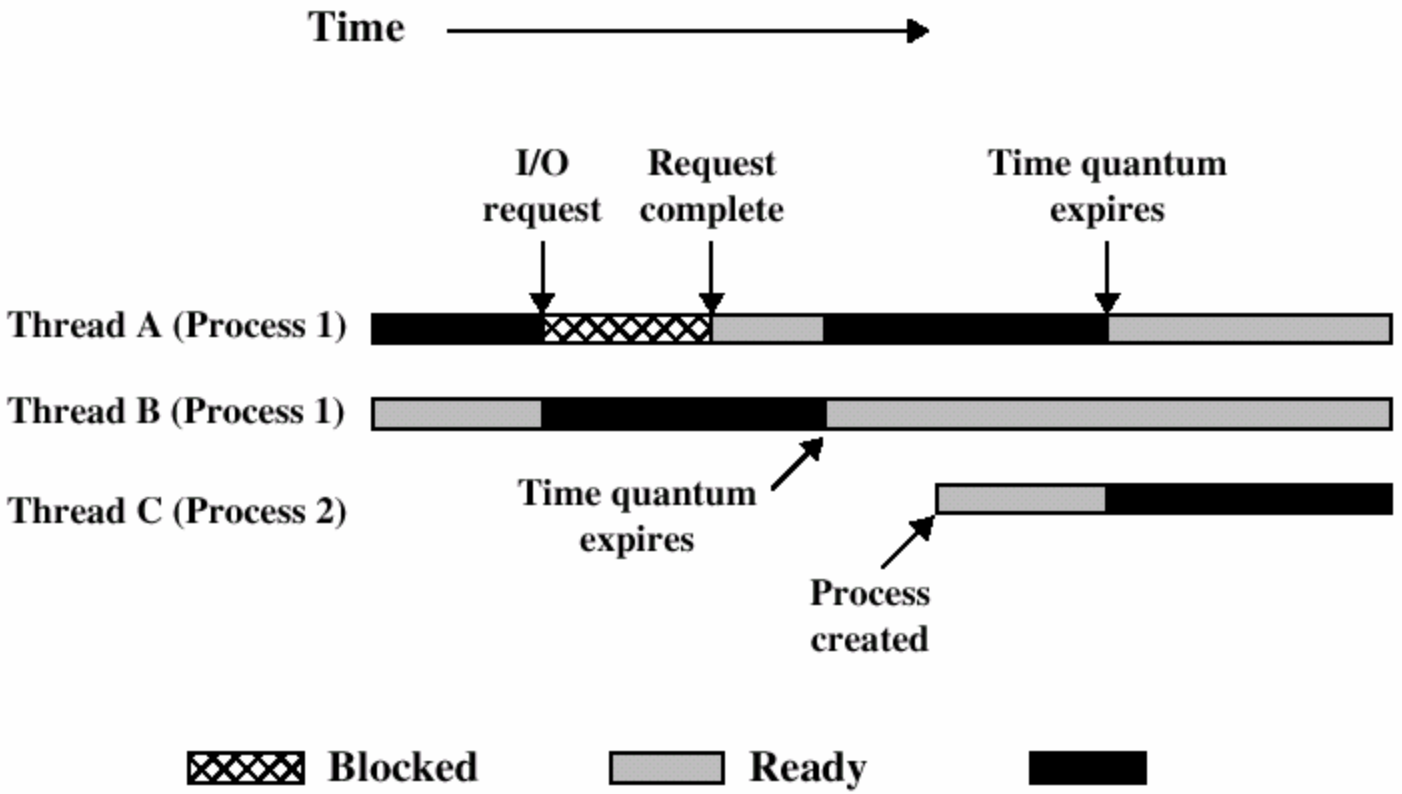
Threads können wg
Blockierungen bei E/A etc
und zur Verbesserung
der Softwarearchitektur auch
bei Einprozessormaschinen
Sinn machen.

Beispiel
Remote Procedure Calls
für unterschiedliche Server
falls Ergebnisse von RPC 1
für Parameter von RPC 2
nicht erforderlich sind
a) mit 1 Thread
b) mit 2 Threads



(Stallings, Fig.4.4)

Multiprogramming & Multithreading



– (Stallings, Fig.4.4)

Multithreading erlaubt es einem Prozeß seltener zu blockieren und damit seine Zeitscheibe besser zu nutzen.

Ein kurzer Ausflug in Threadprogrammierung in Java

Wie sind Threads in gängige Programmiersprachen integriert ?

- C/C++: OS spezifische Bibliotheken, zusätzlich POSIX Standard konforme Bibliotheken mit API zur Threaderzeugung, Synchronization, mit Semaphoren etc. …
- Java: unmittelbar im Sprachumfang enthalten,
  - je nach Virtual Machine: Green Threads, Native Threads

Threads in Java

- durch Vererbung aus der Thread Class
- durch ein Runnable Interface

Lebenszyklus

- Deklaration: Thread t = new Thread(this);
- Initiieren: t.start()
- Ablaufen lassen: run()
- Terminieren -> durch Terminierung von run()

Threads in Java

- Thread Scheduling:
  - yield(): freiwillige Aufgabe des Prozessors
  - getPriority(), setPriority(): Prioritäten zur programmgesteuerten Auswahl eines Threads
- Wechselseitiger Ausschluß:

Ziel des wechselseitigen Ausschlusses ist es, dass jeweils nur 1 Thread zu einem Zeitpunkt den Zugriff auf eine Ressource/ein Datum erhält

Maßnahmen

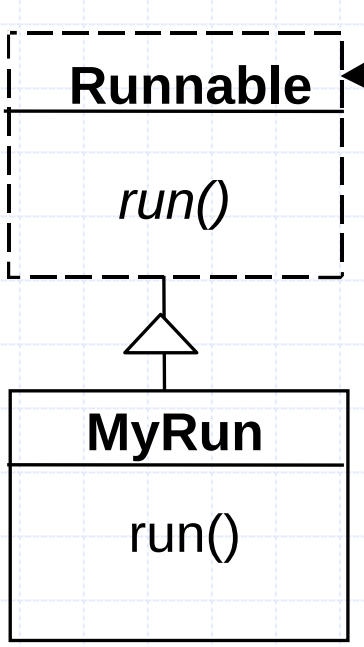
- Synchronized: je Objekt existiert in Java ein Lock (=Sperre), der in einer atomaren Operation gesetzt oder freigegeben werden kann. Java VM sichert anschliessend, dass alle Methoden mit Attribut Synchronized jeweils nur von demjenigen Thread ausgeführt werden können, der den Lock gesetzt hat.
- Wait(): Warten auf Benachrichtigung.
- Notify(), NotifyAll(): Versenden einer Benachrichtigung

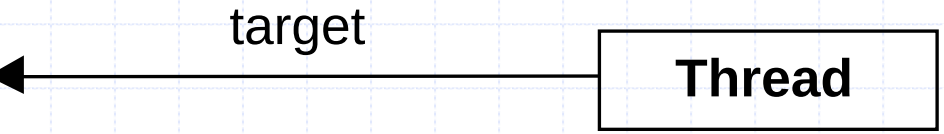
Wait-Notify wird eingesetzt, um eine Reihenfolge in der Bearbeitung von Teilschritten zu erreichen, z.B. falls A vor B, B vor C, C vor D bearbeitet werden muss und 1 Thread A, C ein 2. Thread B und D bearbeitet:

1. Thread: A, notify, wait, C, notify 2. Thread: wait, B, notify, wait, D

Threads in Java

Implementierung eines Interfaces “runnable”.





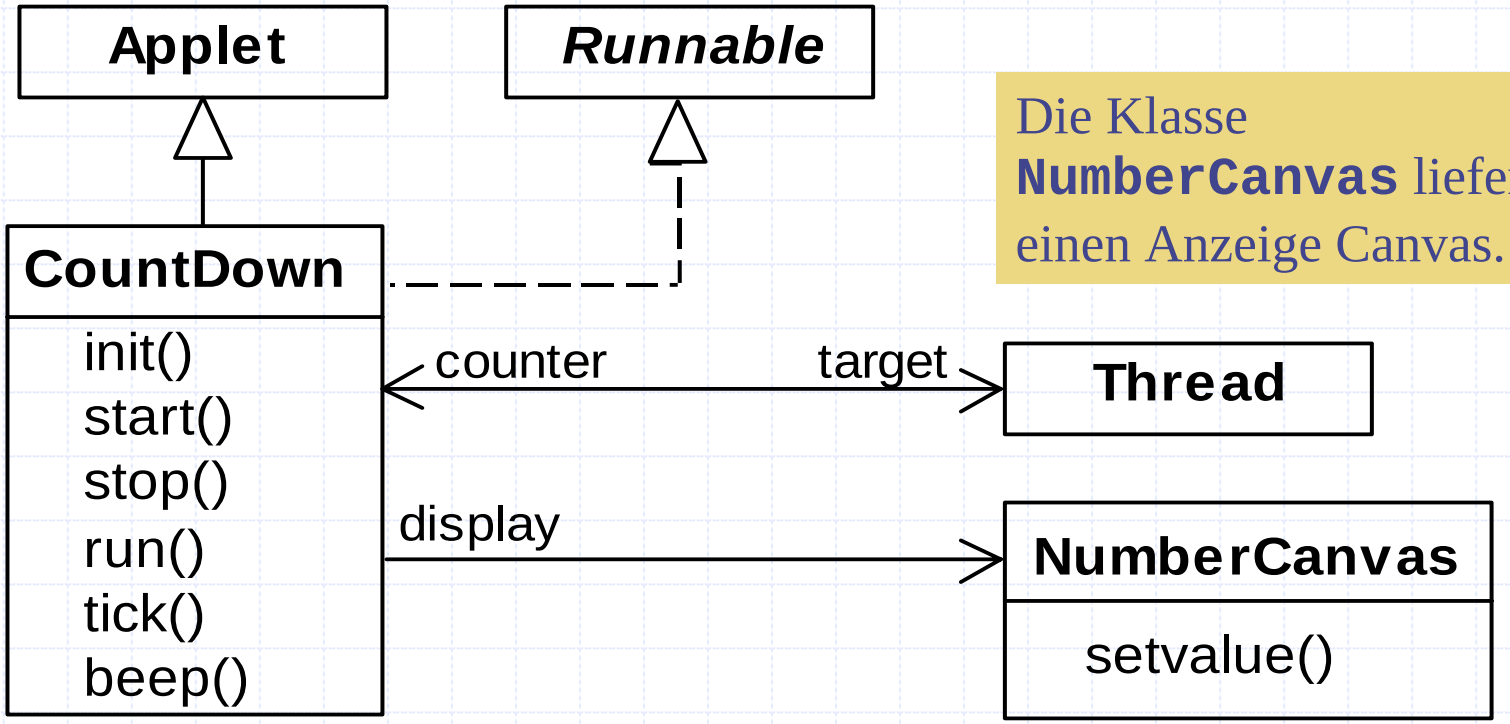

```
public interface Runnable {
    public abstract void run();
}

class MyRun implements Runnable {
    public void run() {
        //.....
    }
}
```

Wie erzeugt MyRun Methode Thread mit eigener run() Methode?

```
Thread lokalMyRun = new Thread(this) ;
```

Beispiel: CountDown timer - Klassendiagramm

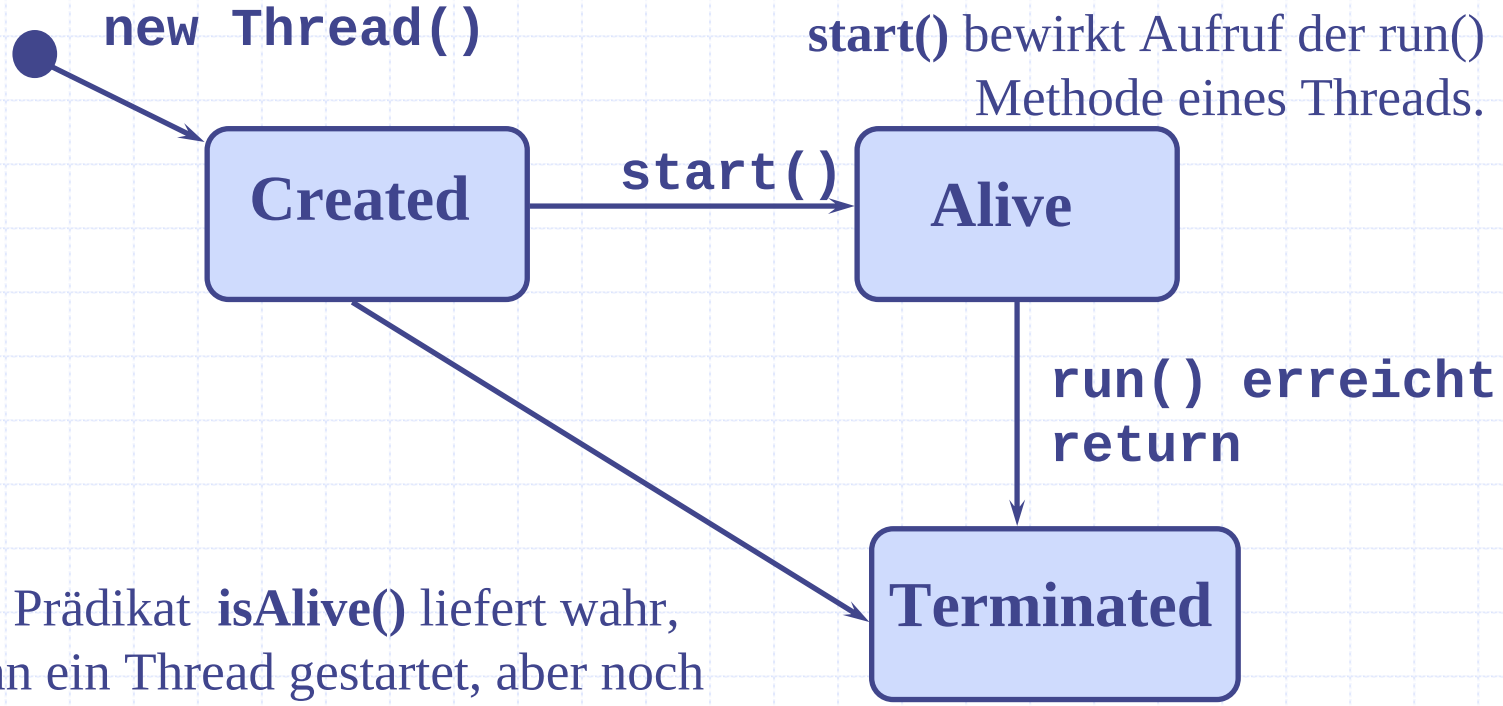


Die Klasse **`NumberCanvas`** liefert einen Anzeige Canvas.

Die Klasse **`CountDown`** erbt aus **`Applet`** und implementiert die **`run()`** Methode für einen Thread.

(Einfacher) Lebenszyklus eines Threads in Java

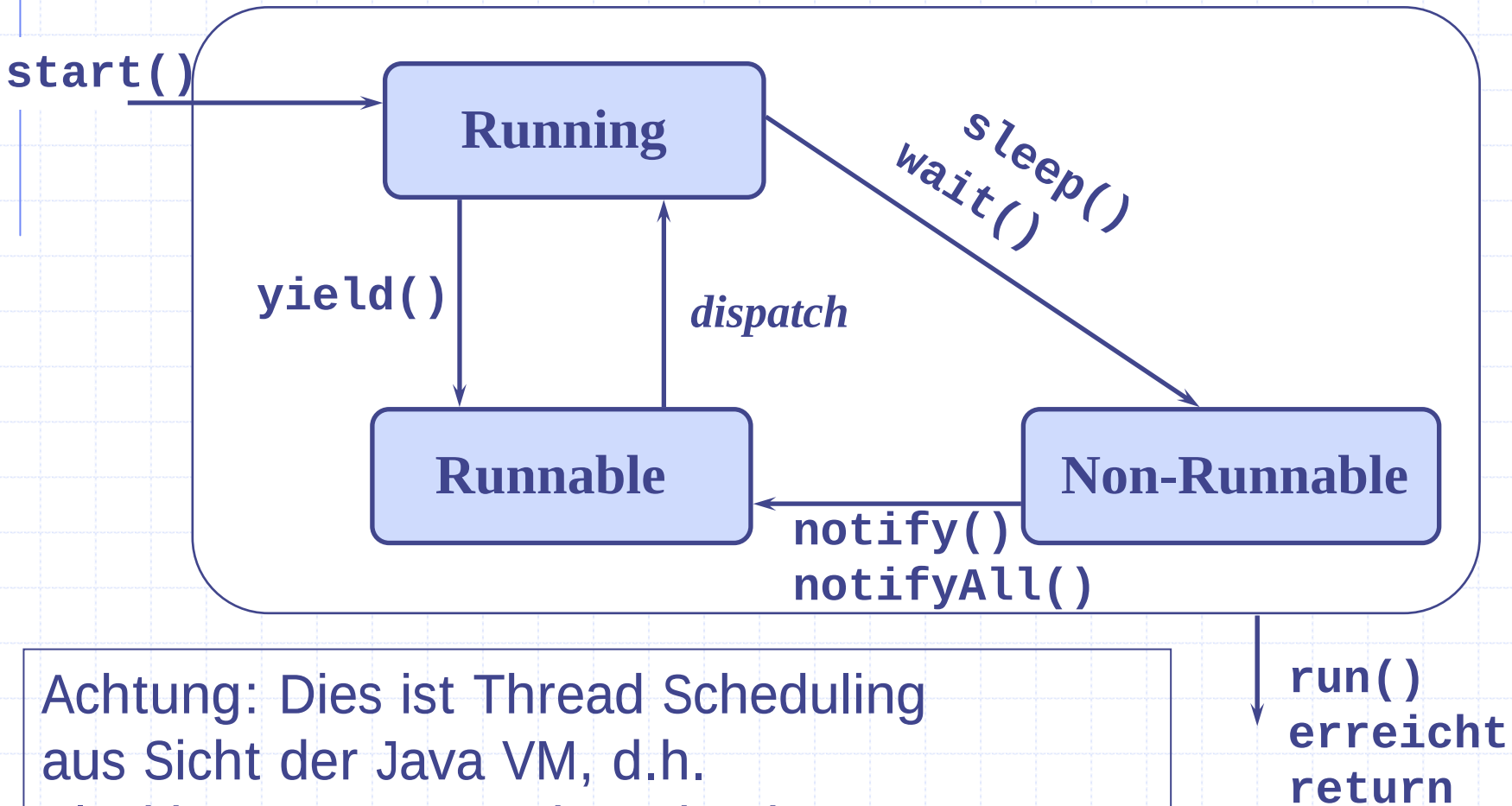
Zustandsübergangssystem zur Darstellung des Lebenszyklusses



Das Prädikat **isAlive()** liefert wahr, wenn ein Thread gestartet, aber noch nicht beendet ist. Einmal terminiert, immer terminiert (kein Restart).

Thread Zustände eines “lebenden” Java Threads

Nach Start und vor Terminierung existieren unterschiedliche Thread Zustände :



Achtung: Dies ist Thread Scheduling aus Sicht der Java VM, d.h. Blockierungen entstehen durch Synchronisation/ Kommunikation der Threads

Initiierung eines Threads: Methode start()

- Achtung nicht mit Applet.start() verwechseln !!!
- MyThread.start() bewirkt das Starten des Threads MyThread, d.h. die run() Methode von MyThread wird aufgerufen.
- Warum daher nicht einfach MyThread.run() ???

  Weitere Schritte erforderlich:

  Initialisierung des Threads innerhalb der VM,
  Integration in Verwaltungsdatenstrukturen für Scheduling, Accounting etc.
- start() darf je Thread nur einmal aufgerufen werden,

  2. Aufruf liefert IllegalThreadStateExecption falls für Thread isAlive() gilt.

Bearbeitung eines Threads, Methode run()

- Run()
  - ist Hauptanforderung zur Implementierung von Runnable
  - ist Hauptfunktion eines Threads
  - wenn Run() terminiert (return), terminiert der Thread
- ab Java 2 erfolgt Threadterminierung im wesentlichen auf freiwilliger Basis d.h. Thread terminiert
  - wenn Thread seine Berechnungen/Aufgaben abgeschlossen hat
  - wenn anderer Thread Aufforderung für Terminierung mitteilt, dies geschieht durch Setzen von eigenen, lokalen Daten oder geteilten Objekten.

    Voraussetzung: Thread kontrolliert Bedingungsflags selbst

Beispiel: CountDown Class

```
public class CountDown extends Applet
                       implements Runnable
{
  Thread counter; int i;
  final static int N = 10;
  AudioClip beepSound, tickSound;
  NumberCanvas display;
  public void init()  {...}
  public void start() {...}
  public void stop()  {...}
  public void run()   {...}
  private void tick() {...}
  private void beep() {...}
}
```

Die Klasse CountDown - start(), stop() und run()

```
public void start() {
   counter = new Thread(this);
   i = N; counter.start();
 }

 public void stop() {
   counter = null;
 }

 public void run() {
   while(true) {
     if (counter == null) return;
     if (i>0)  { tick(); --i; }
     if (i==0) { beep(); return;}
   }
 }
```

Achtung: Namensgleichheit

Thread - Applet

bzgl

start, stop

Hier: Applet erzeugt bei start() einen Thread und startet diesen.

Thread Konstruktor mit Parameter (this)

bewirkt das counter.start()

die Methode this.run()

aufruft.

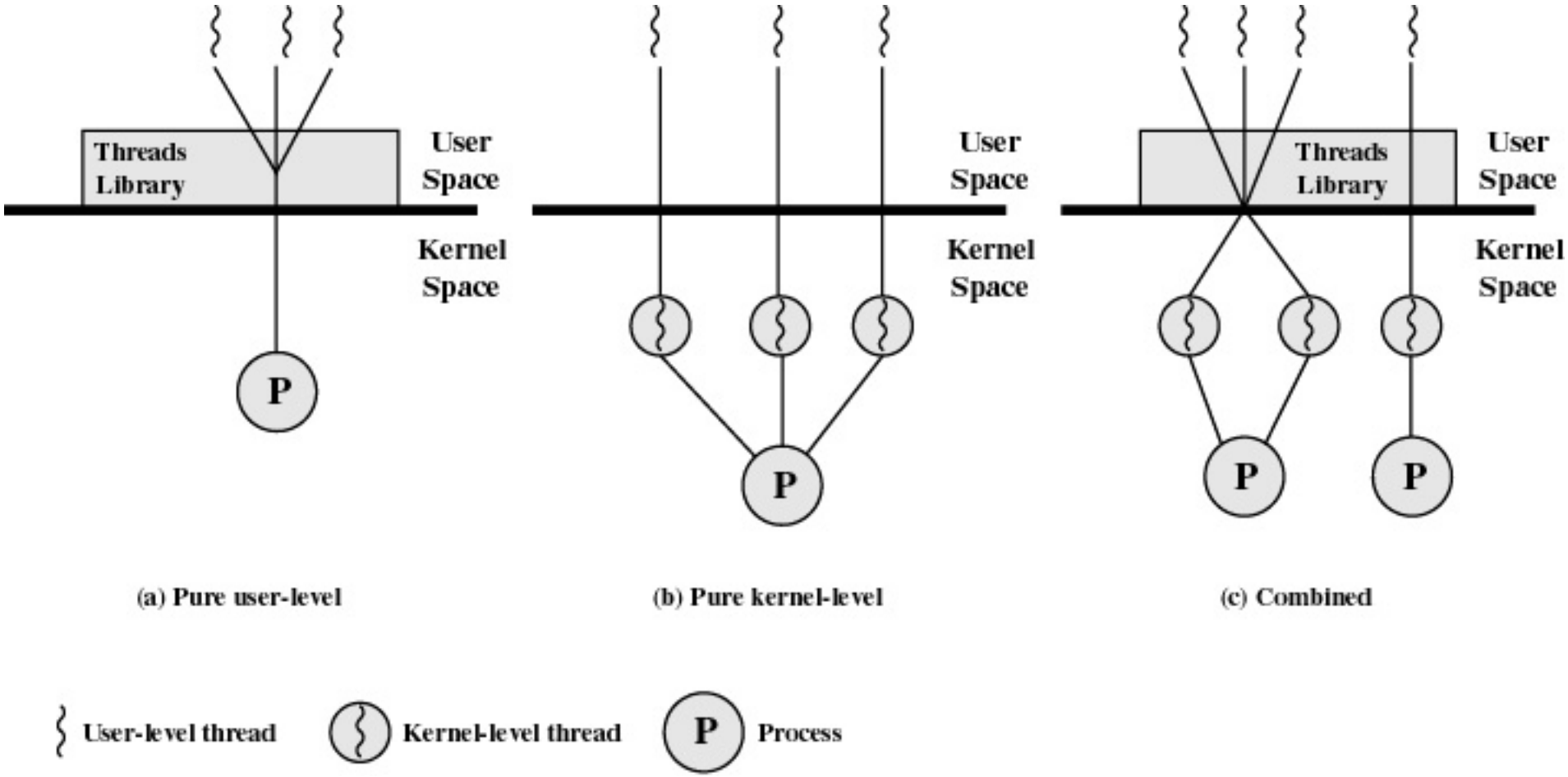
Terminierungserkennung über Bedingungen

counter == NULL !!!

Threads in Java

- Threads durch Vererbung oder Implementierung des Runnable Interfaces
- Methoden zum Initialisieren (start) und Betreiben (run)
- weitere Thread Unterstützung in JAVA:
- Kommunikation/Synchronisation:
  - wait-notify: Warten und Benachrichtigen (Aufwecken)
  - interrupt- wait,sleep,join (ähnliches Prinzip)
- Scheduling
  - yield : freiwillige Aufgabe des Prozessors
  - getPriority, setPriority: Prioritätenvergabe durch Programmierer Java Thread Scheduling der VM arbeitet nach Prioritäten und unterbrechend

Java zeigt, dass Threads auf Anwendungsebene sichtbar sind.
Heißt das, dass Threads ebenso vom OS gesehen werden?
User-level Threads vs Kernel-level Threads



Stallings Fig 4.6: Threads aus OS Sicht, 3 Umsetzungsvarianten

a) OS sieht nur 1 Prozeß, Threads existieren nur innerhalb der Applikation

b) OS sieht n Threads, führt Thread Scheduling durch (Windows 2000, Linux)

c) OS sieht m Kernel Threads = leichtgewichtige Prozesse bei n Threads innerhalb der Applikation (UNIX-Solaris)

Zeitbedarf von Thread / Prozeßoperationen

Messungen auf einer VAX aus dem Jahre 1992,
absolute Zahlen sind eher uninteressant,
Zahlenverhältnisse sind immer noch interessant:

(Stallings, Tab. 4.1, nach [ANDE92])

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

zum Vergleich: Funktionsaufruf 7 μs, Kernel Trap 17 μs
Die Zahlen machen deutlich, dass User-Level Threads kaum Performance kosten und daher bei passenden Aufgabenstellungen für eine klare SW-Architektur gut einsetzbar sind. Kernel-Level Threads liegen um einen Faktor 10-30 höher und machen daher nur zur Vermeidung von Blockierung bei E/A Aufrufen oder bei Mehrprozessorarchitekturen Sinn, auf denen tatsächlich parallel gerechnet werden kann.
Operationen für Prozesse sind um 2 Größenordnungen teurer.
Dies zeigt die Nützlichkeit von Threads ...

User-level Threads vs Kernel-level Threads (ULT vs KLT)

Pro ULT

- Threadwechsel ohne Kernelnutzung, spart 2 x Mode Switch
- Scheduling liegt innerhalb der Anwendung, passend wählbar
- Konzept ist OS unabhängig

Contra ULT

- Systemcalls sind oft blockierend => Prozeß blockiert stets komplett
- kein Multiprocessing auf einer Mehrprozessorarchitektor

Pro KLT

- Kernel leistet Threadscheduling => Multiprocessing bei mehreren CPUs, Prozess muss nicht blockieren wenn 1 Thread blockiert
- Kernel Routinen können selbst mit Threads implementiert sein

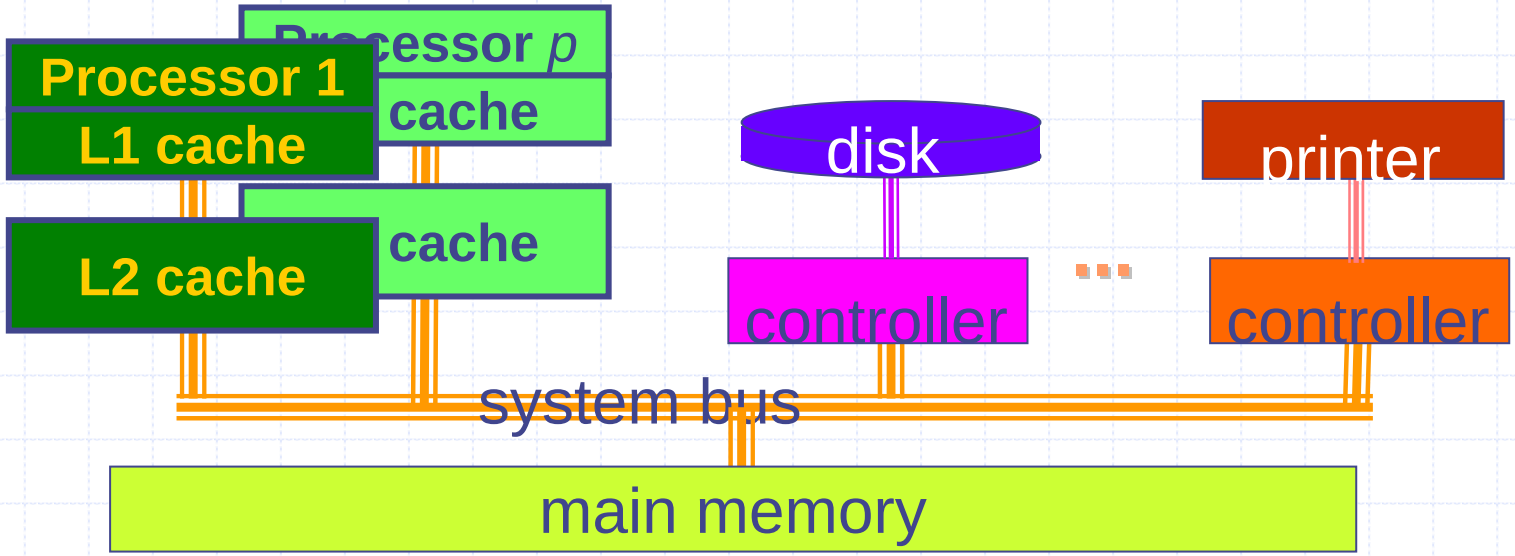
Contra KLT

- Threadwechsel erfordert Modeswitch zum Kernel

Fazit: beides nützlich, daher in praxi Kombination aus beiden, z.B. Solaris, „leichtgewichtige Prozesse“ verbinden ULT&KLT

Symmetrischer Mehrprozessorbetrieb (SMP)

- Rechner mit mehreren CPUs werden nach 2 Mustern betrieben
  - Master/Slave Betrieb: Kernel wird von genau 1 CPU (Master) bearbeitet, alle anderen CPUs bearbeiten Anwendungen (Slaves)
  - Symmetrischer Mehrprozessorbetrieb: Kernel wird von allen CPUs je nach Bedarf bearbeitet, jede CPU betreibt eigenes Scheduling und wählt aus Pool bearbeitbarer Prozesse / Threads aus. Betriebssystem-Code wird dadurch teilweise parallel bearbeitet.



SMP

Schwierigkeiten durch Parallelität:

- OS selbst muss auch bei paralleler Durchführung von Kernelinternen Aufgaben sicher funktionieren
- Konsistenz von Cache Inhalten in den CPU internen Caches
- Exklusive Zuordnung von Prozessen, Threads und Ressourcen,
- Kommunikation zwischen CPUs via Speicher oder Nachrichten

Anforderungen an SMP Betriebssystem

- Simultane Berechnung von parallelen Prozessen & Threads (reentrant Code, keine Interferenz bei Zugriffen, ...)
- Scheduling: verteiltes Multiprozessorscheduling muß auch Verteilung von Threads auf mehrere Prozessoren ermöglichen
- Synchronisation beim Zugriff auf geteilte Ressourcen ermöglichen
- Speicherverwaltung: Koordination der Seitenanforderungen der einzelnen CPUs, insbesondere bei geteilten Speicherbereichen
- Zuverlässigkeit und Fehlertoleranz: „graceful degradation“ bei Ausfall einer CPU

Fazit: zusätzliche Schwierigkeiten im Vgl zu Einprozessor OS, daher zunächst Lösungen für grundsätzliche Probleme auf Einprozessorrechnern

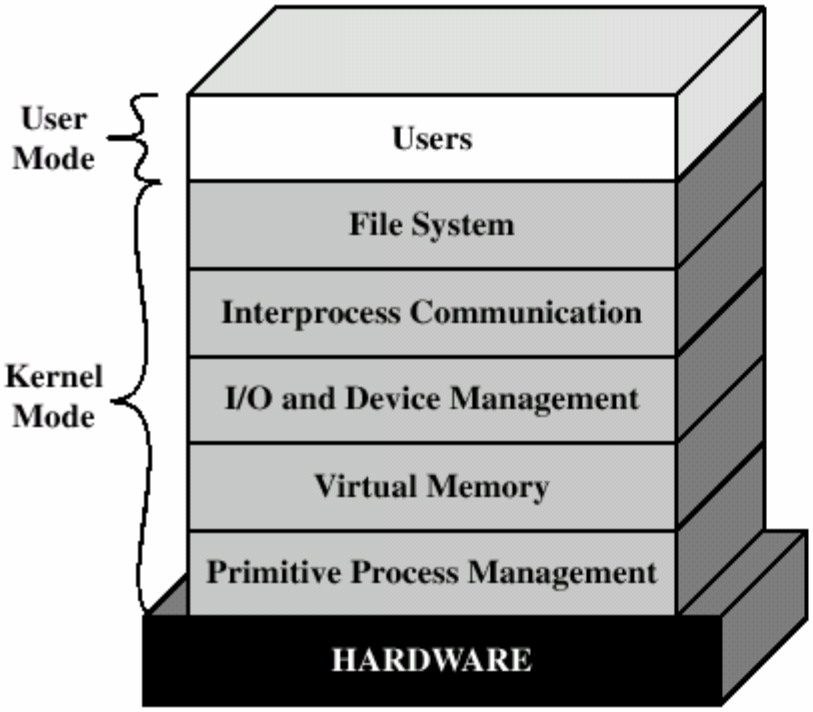
Microkernel

OS als Software komplex, daher historische Entwicklung:

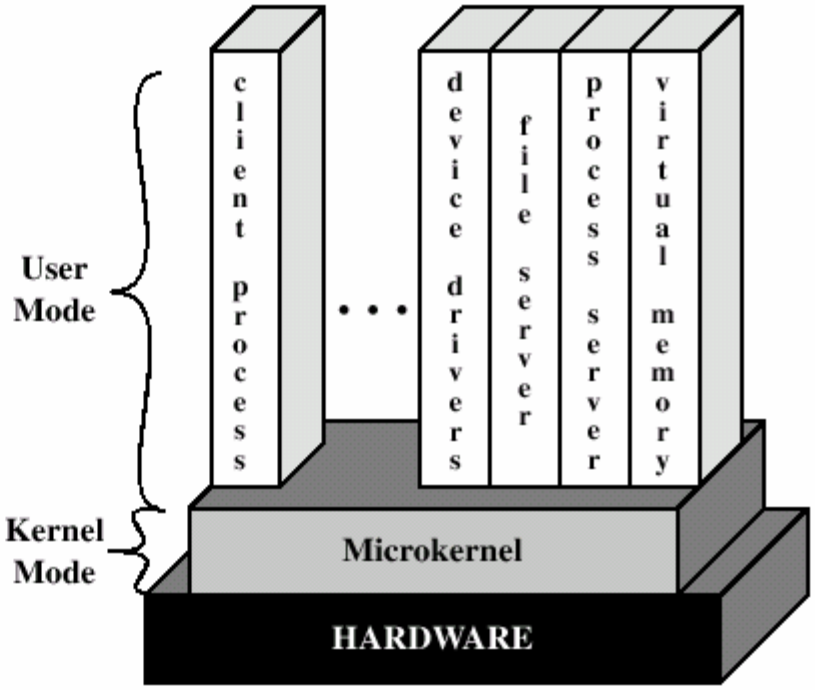
monolithisch->monolithischer Kernel->Schichtenarchitektur

->Microkernel

Idee: Kernel umfasst nur ganz wenige Funktionen, kapselt Hardware und vermittelt Nachrichten zwischen Anwendungsprozessen und OS-Dienstprozessen außerhalb des Kernels.



(a) Layered kernel



(b) Microkernel

Microkernel

Sichtweise: OS ist ein System kommunizierender Prozesse,
die Dienstleistungen erbringen,
und ein Vermittlungssystem für Nachrichten zwischen Prozessen.

Mögliche Vorteile:

- Einheitliche Schnittstellen
- Erweiterbarkeit und Flexibilität im Funktionsumfang
- Portabilität
- Zuverlässigkeit
- Unterstützung für verteilte Anwendungen & Systeme
- Unterstützung für ein objektorientiertes OS

Mögliche Nachteile:

- Performance
  - Ausweg 1: Microkernel vergrößern, so dass besonders kritische oder ausschlaggebende Aufgaben doch im Kernel liegen (MACH, CHORUS)
  - Ausweg 2: Microkernel verkleinern, besseres Design zahlt sich evtl aus (L4 hat 12Kbyte und 7 Systemaufrufe)

Aufgabenumfang eines Microkernels

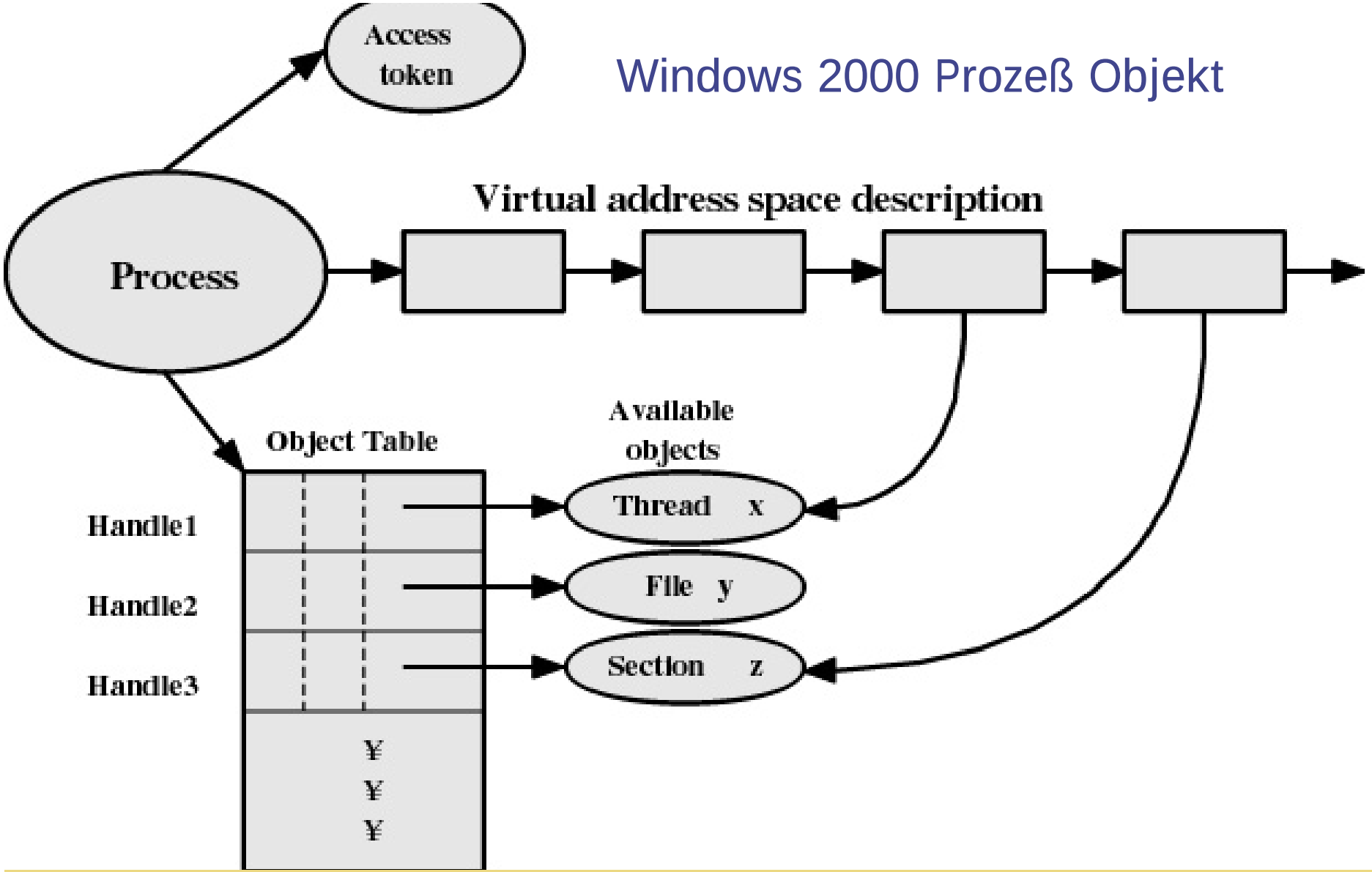
- Speicherverwaltung auf Maschinenebene
  - Microkernel realisiert nur die Zuordnung von virtuellen Adressen zu physikalischen für Speicherzugriffe und leistet damit Speicherschutz. Aufgaben wie die Seiten- oder Segmentverwaltung erfolgt außerhalb des Kernels.
  - wesentliche Funktionen: Zugriffsrechte geben (grant) und zurücknehmen (flush), Speicherbereiche zwischen Prozessen überlappen lassen (map)
- Interprozeßkommunikation mittels Nachrichten
  - Annahmestelle eines Prozesses: Port
  - bei Prozessen mit unterschiedlichen Adressräumen

    => Kopieren von Daten im Speicher (Geschwindigkeit)
- E/A und Interrupt Management
  - wegen einer einheitlichen Behandlung von Nachrichten und Interrupts werden Interrupts durch den Kernel in Nachrichten umgewandelt und an passende Prozesse / Threads weitergeleitet
  - E/A Geräte erhalten ebenfalls Ports und werden als Threads betrachtet

Threads in realen Betriebssystemen

- Windows 2000
  - wg der Emulation von diversen Betriebssystemen (Win32, OS/2, ...), einfache aber generelle Struktur für Prozesse
  - Prozesse sind Objekte, können mehrere Threads enthalten
  - Thread und Prozeß Objekte unterstützen Synchronisation
- Solaris
  - unterscheidet User-Level Threads und Kernel-Level Threads
  - KLTs sind Ressourcen und werden ULTs mittels leichtgewichtiger Prozesse (LWPs) zugeordnet.
  - Heuristik: ein Prozess mit ULTs bekommt jeweils einen weiteren LWP, wenn der aktive ULT blockiert, aber noch ULTs bereit sind.
- Linux
  - keine echte Unterscheidung von Threads und Prozessen, Threads werden als Prozesse mit geteilten Speicherbereichen realisiert
  - Threaderzeugung durch Klonen, d.h. Erzeugen eines Prozesses mit gleicher Ressourcenzuordnung (auch Speicher) und gleichen Attributen

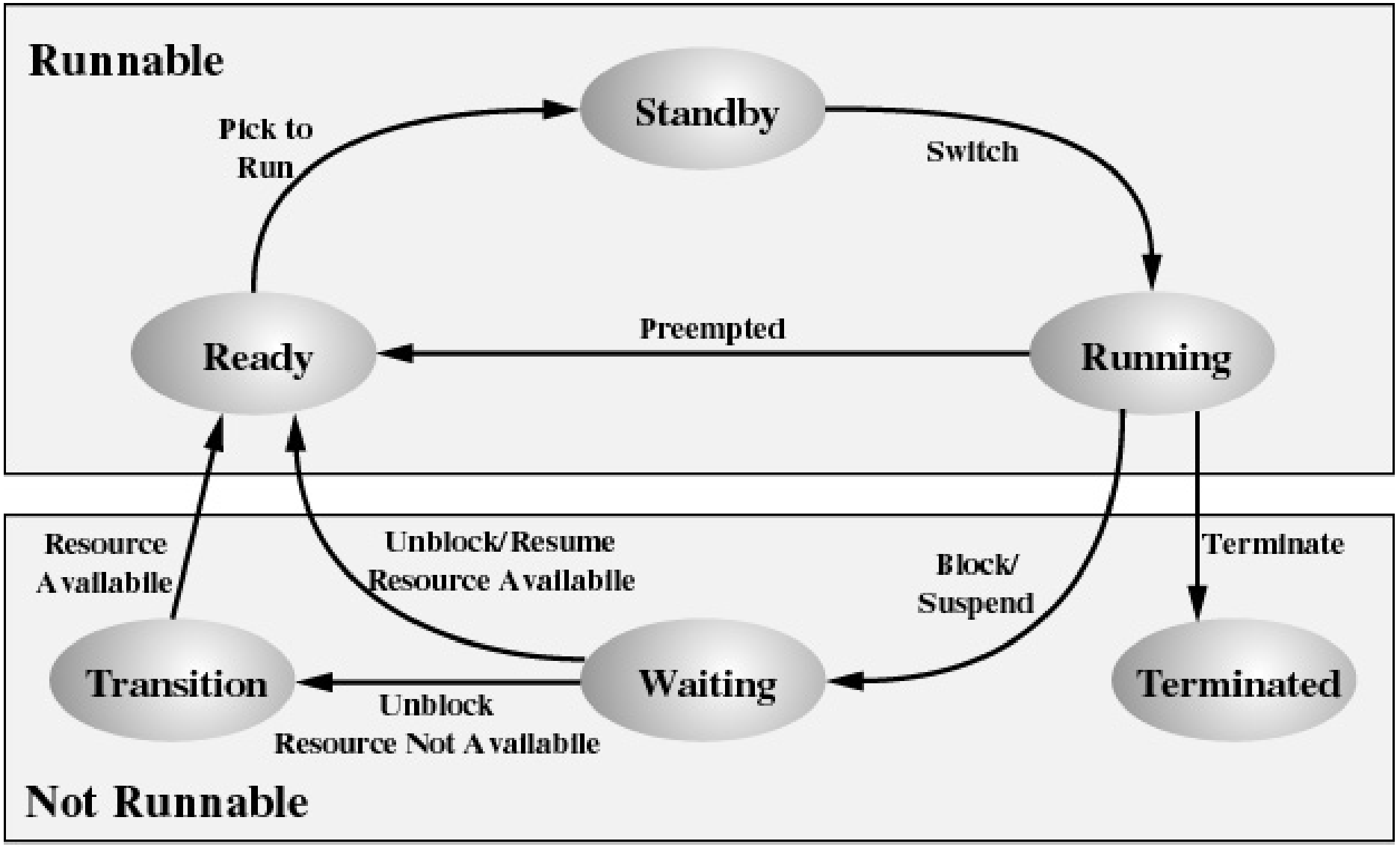
Windows 2000 Prozeß Objekt



Stallings Fig 4.12: Windows 2000 :Datenstruktur zur Abbildung eines Prozesses, Prozesse & Threads werden als Objekte realisiert, Tabelle mit Handles = Griffen beschreiben zugeordnete Objekte

Windows 2000 - Welche Zustände erfährt ein Thread ?

- Ready:
  - zur Ausführung bereit. Der Microkernel Dispatcher berücksichtigt alle bereiten Threads und ordnet sie nach Prioritäten.
- Standby:
  - zur Bearbeitung für einen bestimmten Prozessor als nächster Thread bestimmt. Falls Priorität ausreicht wird der aktive Thread des Prozessors unterbrochen (preemption), ansonsten muß der Standby Thread auf eine Blockierung oder das Ende der Zeitscheibe warten.
- Running:
  - Thread wird bearbeitet, bis Terminierung, Blockierung, Unterbrechung oder das Ende der Zeitscheibe erreicht wird.
- Waiting, auf eine Bedingung warten:
  - wird durch Blockierung wg eines Events (E/A), Synchronisation oder einer Anweisung eines anderen Threads bewirkt. Falls die Bedingung erfüllt ist, wird der Thread Ready (sofern Ressourcen verfügbar sind).
- Transition:
  - Thread ist eigentlich ready aber Ressourcen fehlen (z.B. Speicherseiten).
- Terminated
  - kann durch den Thread selbst, andere Threads oder Terminierung des Elterprozesses bewirkt werden, führt zur Speicherfreigabe.



Stallings 4.14: Windows 2000: Zustandsdiagramm für Threads

„Transition“ Zustand: Ready aber Ressourcen fehlen, zB wg Paging

„Standby“: bereits ausgewählter Nachfolger

Zusammenfassung

- Threads in Java
  - Übersicht: Definition und Methoden
  - start(), run(), wait(), notify(), notifyAll()
  - Scheduling
- 3 Varianten für die Behandlung von Threads in OS
  - a) User-Level Threads: applikationsintern
  - b) Kernel-Level Threads: für OS sichtbar und behandelbar
  - c) ULT & KLT, gemischter Ansatz (Solaris: leichtgewichtige Prozesse)
- Symmetrischer Mehrprozessorbetrieb (SMP)
  - CPUs werden gleichmäßig für Kernel und Anwendungen genutzt
  - Echte parallele Bearbeitung von Anwendungen möglich
  - Threads erlauben Performancegewinne
- Microkernel Architektur
  - Grundidee: OS als System kommunizierender Prozesse und elementarem Vermittlungssystem realisiert. Microkernel verdeckt Hardware, vermittelt Nachrichten, bildet virtuelle Adressräume auf physikalische ab.