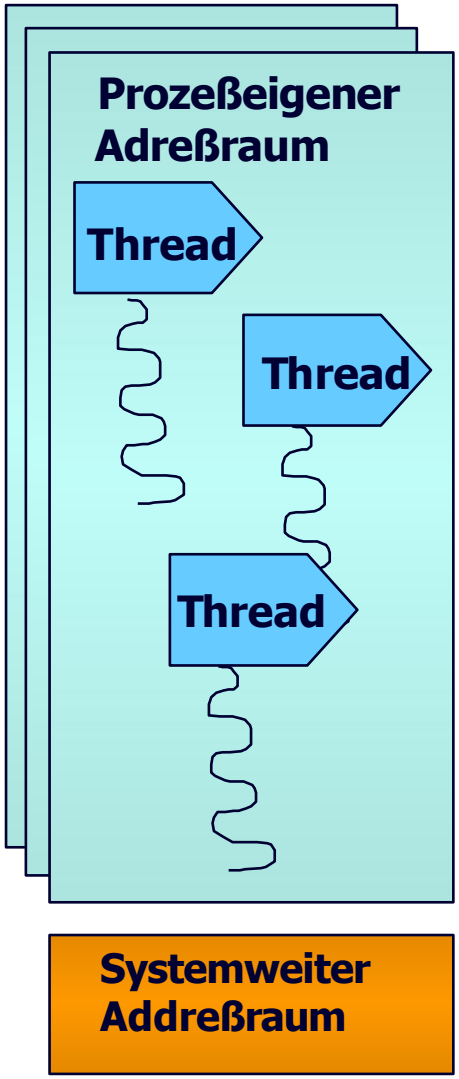
DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Foliensatz „Threads und Prozesse“
Betriebssysteme
V2

Sysinternals Process Explorer (C(ontext)Switches/sec):
https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer

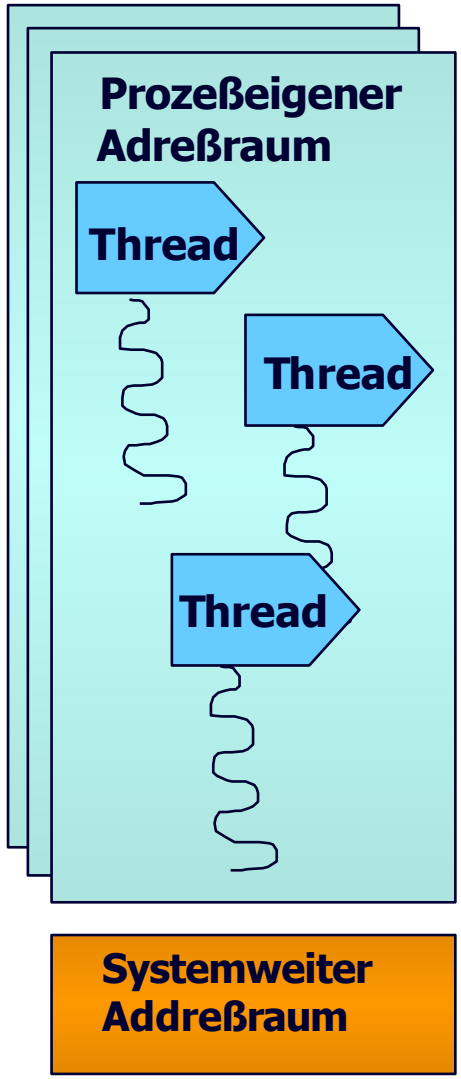
Leichtgewichtige Threads im Prozess

- (Quasi-) Parallele “Fäden” innerhalb eines Prozesses, mindestens 1 Thread ist immer aktiv
- Threads werden softwarebasiert (explizit o. vom Compiler/ Laufzeitumgebung) modelliert
- **Scheduling der Threads** erfolgt innerhalb des Prozesses
  - Register-Inhalt wird umgeschaltet
  - eigener Stack
  - Program Counter: Inhalt wird umgeschaltet



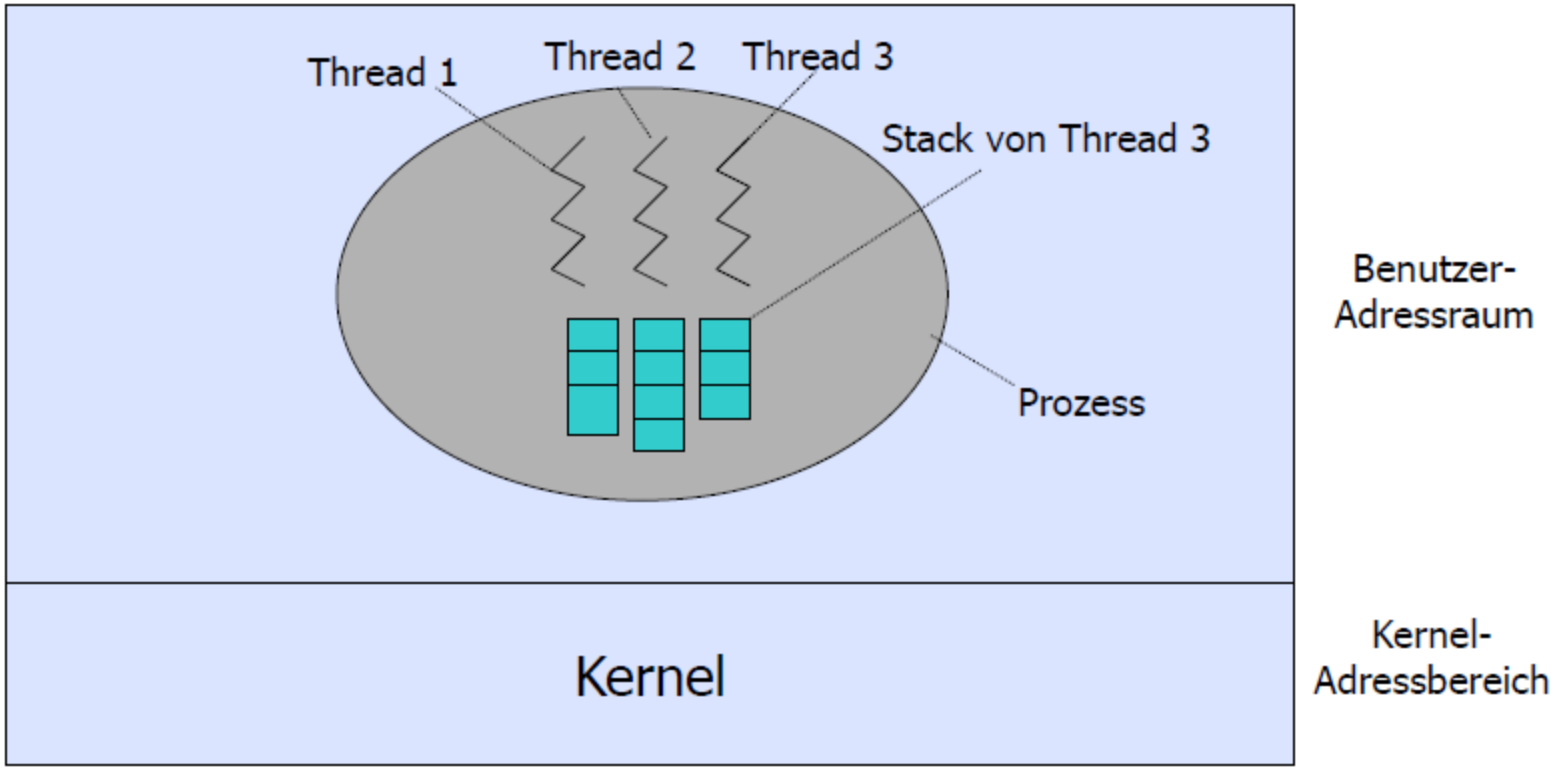
Leichtgewichtige Threads im Prozess

- Die Threads eines Prozesses teilen sich dessen Ressourcen (auch den Adressraum des Prozesses, der nur zwischen den Prozessen abgegrenzt ist), sind gegeneinander nicht geschützt
- beim Threadwechsel ist kein vollständiger Wechsel des Prozesskontextes notwendig
- die Threads können nur laufen, wenn ihr Prozess aktiv ist (alle Threads eines Prozesses laufen auf dem gleichen Prozessor, können aber mehrere Cores der CPU nutzen)



Leichtgewichtige Threads im Prozess

Threads haben einen eigenen Program Counter, einen eigenen log. Registersatz und einen eigenen Stack

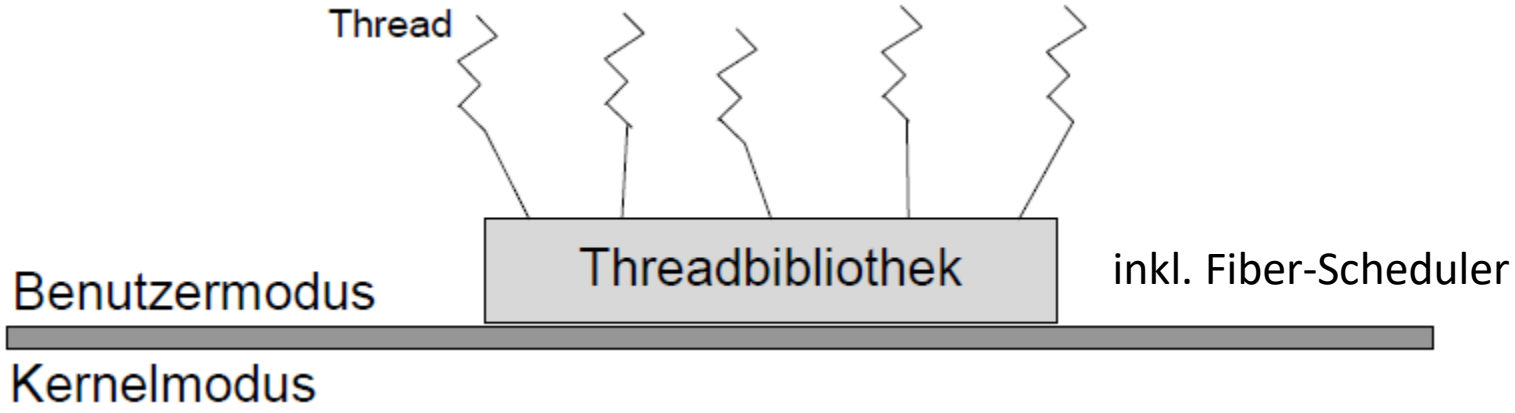


Quelle: *Tanenbaum, A. S.*: Moderne Betriebssysteme, 3. aktualisierte Auflage, Pearson Studium, 2009

Implementierungsvarianten von Threads

1) Implementierung und Scheduling **auf Applikationsebene**
(User-Threads = Fiber-Threads)

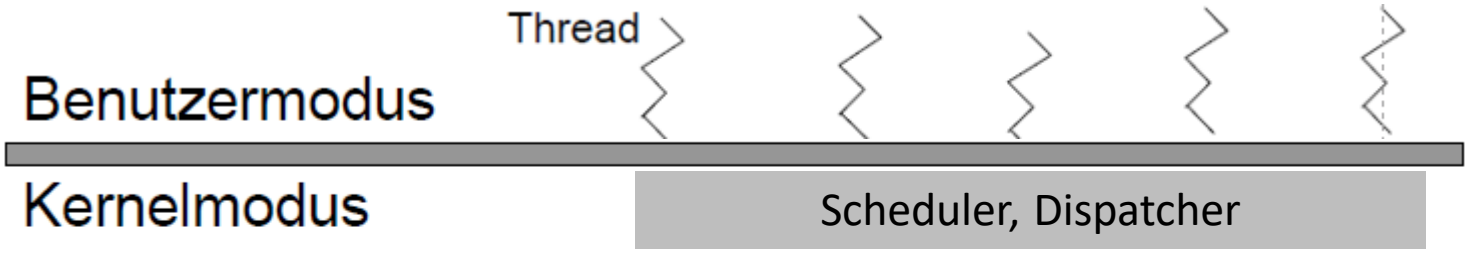
- Thread-Bibliothek übernimmt das Scheduling und Dispatching für Fibers im User-Mode innerhalb von Threads
- Kernel merkt nichts von Fiber-Threads –> OS-unabhängig
- Linux: Bibliotheken (**LinuxThreads)** und **GNUPortableThreads**
- Microsoft: **Fibers** (dünne, einfache Fasern innerhalb eines Threads)



Implementierungsvarianten von Threads

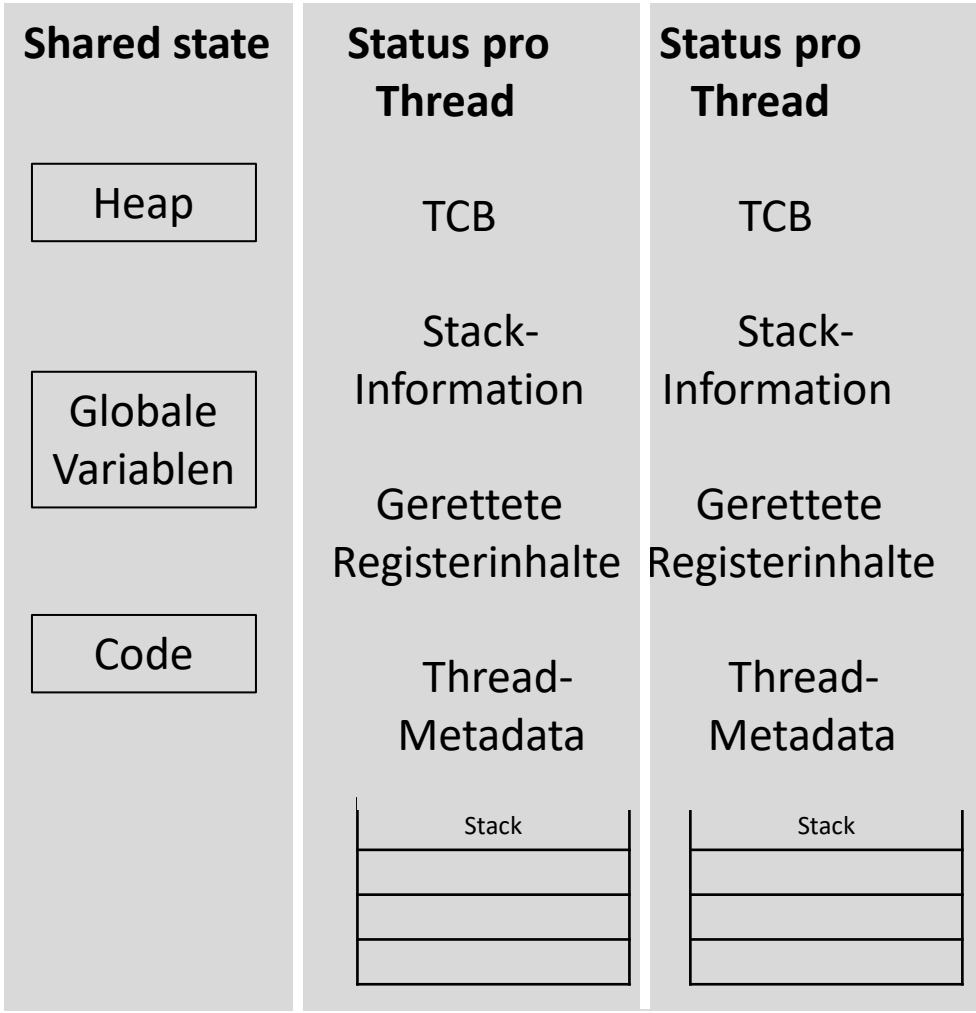
2) Scheduling **auf Kernelebene („Kernel scheduled threads“)**

- Prozess ist nur noch Verwaltungseinheit für Betriebsmittel
- Scheduling-Einheit ist hier der Thread, nicht der Prozess
- Nicht so effizient, da Thread-Kontextwechsel über Systemcall (betriebssystemabhängig)
- Inklusive der vom BS selbst erzeugten Kernelthreads



Kernel (-scheduled) - Threads: Umschaltung

- **Thread Control Blocks (TCBs)** repräsentieren Threads eines Prozesses.

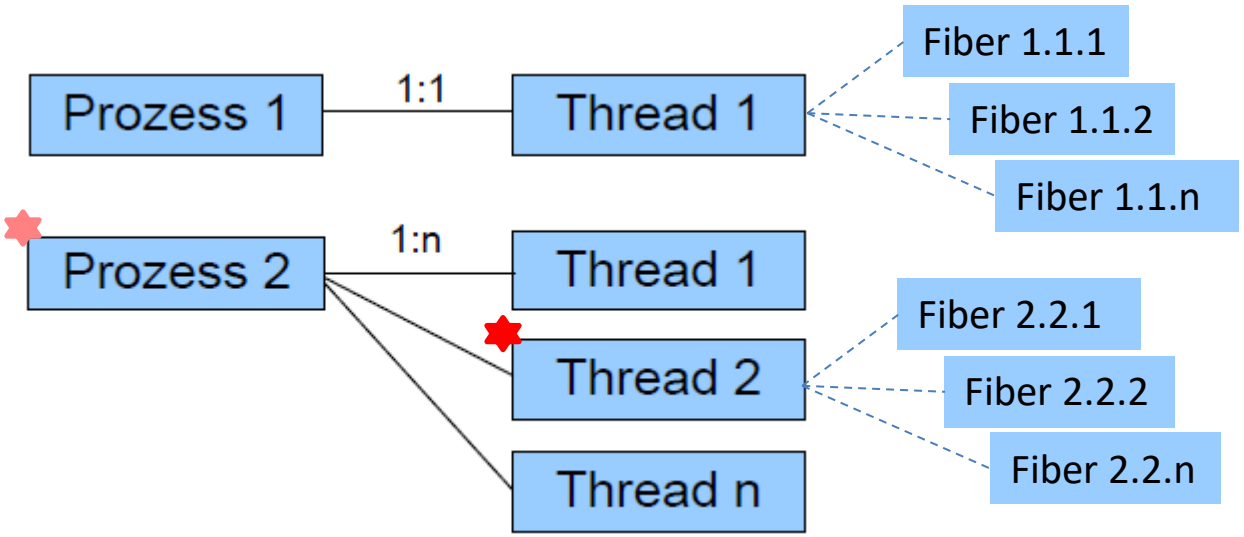


TCB pro Thread

Thread-ID
Thread-Status
CPU-Information: PC, Registerinhalte, SP
Threadpriorität
Pointer auf den Erzeugerprozeß
Pointer auf Kindthread(s)

Kernel(-scheduled) - Threads: Zuordnung zum Prozess

- 1:1 : Genau ein Thread läuft in einem Prozess
- 1:n : Mehrere Threads laufen in einem Prozess (n>0!)



Es muss definiert sein: Was ist die Scheduling-Einheit des OS (✸)? Thread oder Prozess?

Prozess-/Threadverwaltung unter Windows

Job = Gruppe von Prozessen, die bzgl. Ressourcenzuteilung als eine Einheit verwaltet (aber nicht gescheduled) werden, haben Quotas und Limits
- Maximale Speichernutzung je Prozess
- Maximale Anzahl an Prozessen
- ...

Prozess = Container zur Speicherung/ Verwaltung von Ressourcen
- Threads, Speicher, ...

Thread = Scheduling-Einheit – Aktivierung bedingt auch Prozess-Aktivierung

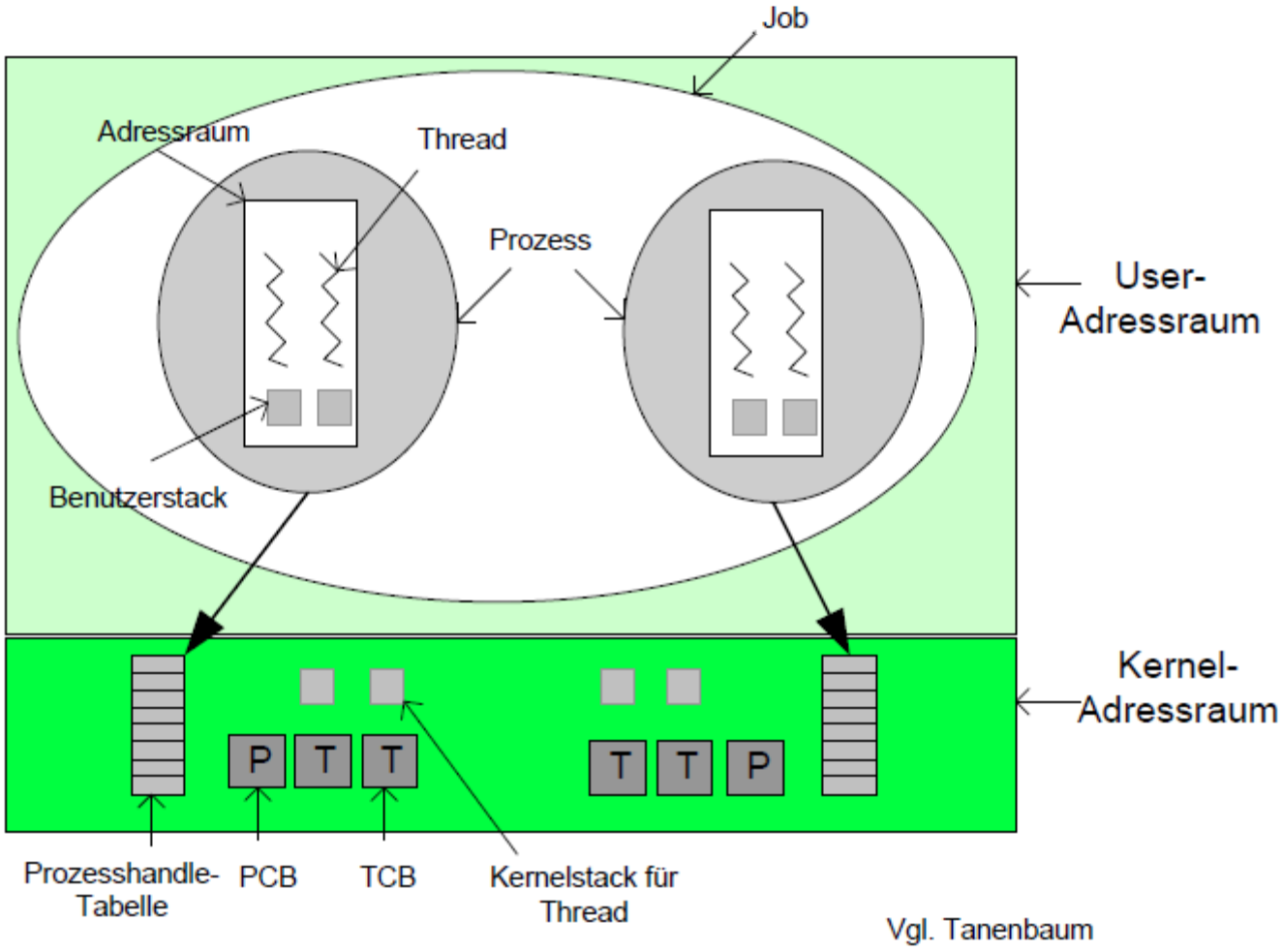
Fiber = Leichtgewichtiger Thread, der von der Applikation (Framework) verwaltet wird (CreateFiber, SwitchToFiber)

*"… Windows schedules at the **thread granularity**. This approach makes sense when you consider that **processes don't run** but only **provide resources and a context** in which their threads run. Because **scheduling decisions are made strictly on a thread basis**, no consideration is given to what process the thread belongs to. …"*

Windows Internals, 5th Edition:
Processes, Threads, and Jobs in the Windows Operating System

Threadverwaltung unter Windows

Jobs, Prozesse und Threads:



Vgl. Tanenbaum

Prozess-/Threadverwaltung unter Unix/ Linux

Job: -

Task/ Prozess = Scheduling-Einheit (UNIX), hat mind. 1 Thread (Unix: genau 1)

Prozesserzeugung mit fork():

```
fork ( SIGCHLD, 0); // sende an den Vater beim Beenden das Signal SIGCHLD
```

Task/ Thread = Scheduling-Einheit (Linux, pThreads (Posix)),
Light weight process (LWP), eigene TID, PC, Stack, Registersatz,
gemeinsamer Adressraum, Erzeugung mit clone() statt fork():

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

Arbeitet wie normale Prozesserzeugung (fork()), aber der Adressraum (VM), Filesystem-Information (FS), offene Files(FILES) und Signalhandler werden geteilt.

Fiber = Leichtgewichtiger Thread, der von der Applikation verwaltet wird

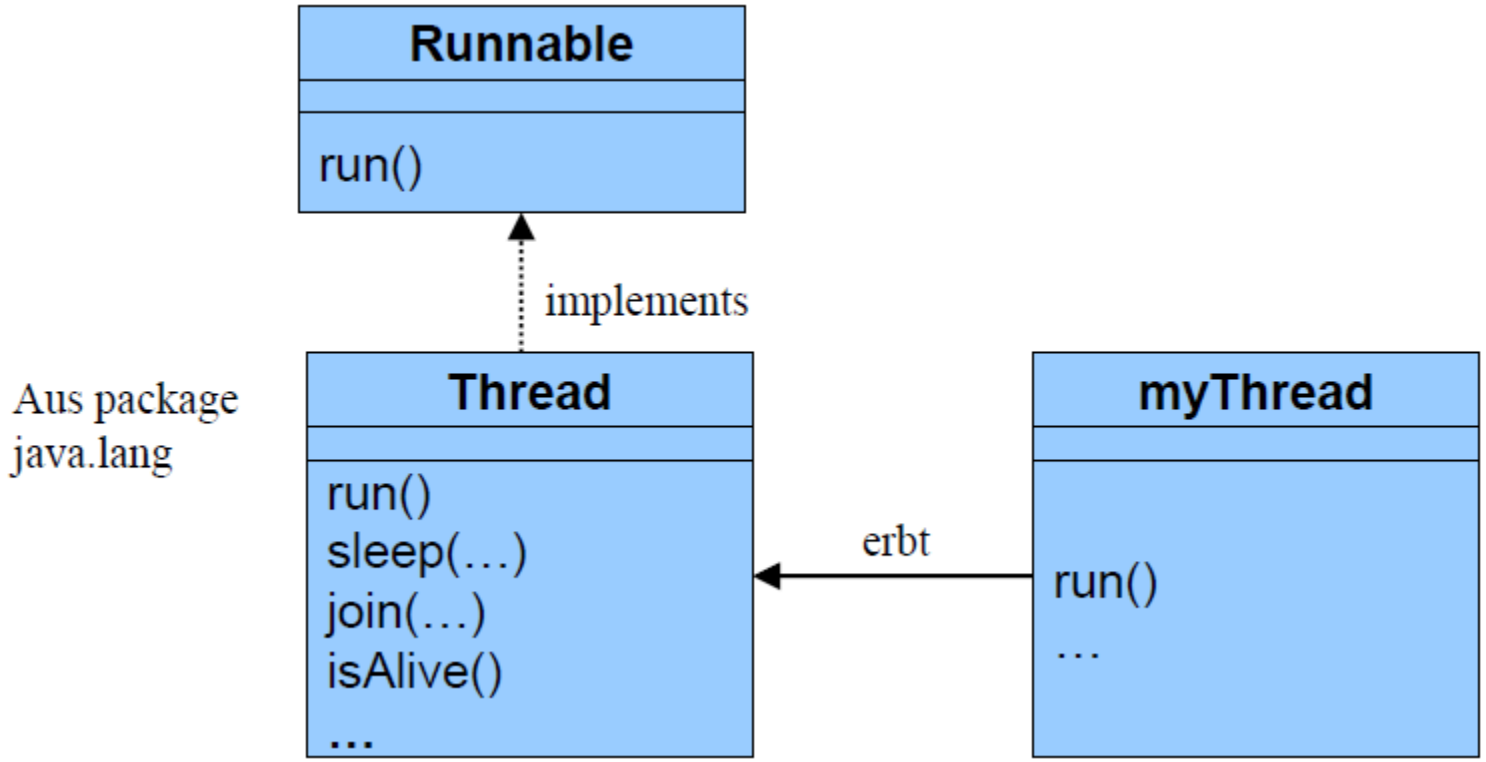
Threads in Java/ JVM

Für jedes Programm wird eine eigene JVM gestartet

- JVM läuft in einem eigenen Betriebssystemprozess
  - Siehe z.B. im Windows Task Manager
- JVM unterstützt OS-/Kernel-scheduled-Threads („native threads“), benutzt dafür das zugrundeliegende OS
- Package **java.lang**
- Basisklasse **Thread**
- „green threads“: Fibers in der JVM, scheduled von JVM, nicht mehr unterstützt

Java-System(-scheduled-)threads in Java/ JVM

- 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



Weitere System (-scheduled-) threads in Java

- Garbage Collection: hat sehr niedrige Priorität (niedriger als Java-Idle-Thread, wartet auf Signal von diesem Idle-Thread)

- Java-Idle: Wenn er läuft, setzt er ein Kennzeichen, das der Garbage-Collector-Thread als Startsignal betrachtet, um etwas zu tun

  -> Idle-Thread wird nur aktiv, wenn die JVM sonst nichts zu tun hat

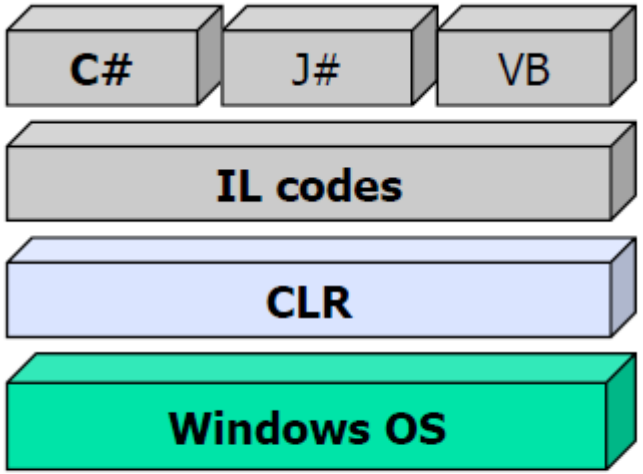
  -> wie zuverlässig läuft der Garbage collector bei hochfrequenter Objekt-Allokation und –freigabe, wenn der Speicher zunehmend fragmentiert?

Threads in C# .NET Framework: CIL, CLR, FCL

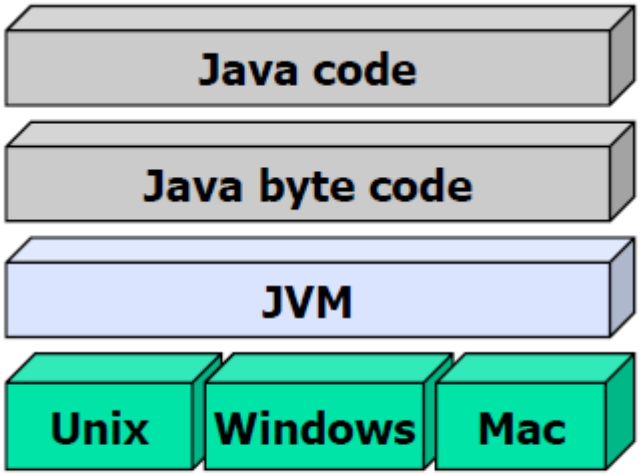
- .NET Framework: Plattform zur Entwicklung und Ausführung von Anwendungs-programmen
- CIL = Common Intermediate Language ist ein Zwischencode
  - entspricht Java Byte Code
- CLR = Common Language Runtime
  - entspricht JVM
- Alle Microsoft-Compiler erzeugen CIL-Code
- FCL = Framework Class Library
  - Klassenbibliothek mit vielen Basisklassen
  - In Namespaces geordnet

Klasse Thread erstellt kernel-scheduled Threads!

Threads in C#: CLR versus JVM



.NET - Lösung



Java - Lösung

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Hyperthreading – Parallelisierung durch Hardware (Intel)

- Hardwareplattform mit mehreren Kernen/ CPUs:
  Hardware selbst nutzt Möglichkeiten der Parallelisierung der Abarbeitung zur besseren Ausnutzung des Kerns, z.B. durch Vorspiegelung mehrerer Kerne pro realem Kern

- Compiler (Intel-Compiler und die GNU Compiler Collection) müssen „hyperthreading-freundlichen Code“ (unter Erkennung und Markierung von parallelisierbaren Abschnitten als Threads) erzeugen

- Berechnungen der Prozesse müssen dafür möglichst parallelisierbar sein (möglichst unabhängig voneinander)

- Mehrere vom Kernel verwaltete Threads werden automatisch auf mehrere (virtuelle) CPUs verteilt

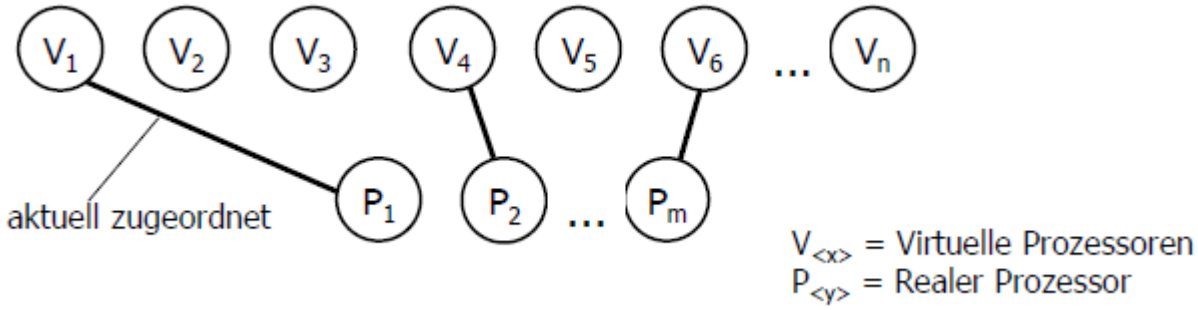
Prozeß und (virtuelle) Prozessoren

Ein **Prozess** (lat. *procedere* = voranschreiten), ist die Ausführung einer Instanz eines Programms durch einen Prozessor

- Prozesse sind dynamische Objekte und repräsentieren sequentielle Aktivitäten im Computer
- Auf Computern sind meist immer mehrere Prozesse (evtl. quasiparallel) in Ausführung
- Die CPU wird bei Multitasking/ -processing im raschen Wechsel zwischen den Tasks der Prozesse hin- und hergeschaltet

Das Betriebssystem ordnet im Multiprogramming jedem Prozess einen (**virtuellen) Prozessor/ CPU** zu:

- Echte Parallelarbeit, falls jedem virtuellen Prozessor ein **realer Prozessor** bzw. Rechnerkern zugeordnet wird
- **Quasiparallel**: Jeder reale Prozessor ist zu einer Zeit immer nur einem virtuellen Prozessor zugeordnet und es gibt Prozess-Umschaltungen



Prozeß und Prozeßkontext

- Ein Prozess umfasst außer dem Programmcode noch seinen **Kontext**
- 3 Arten von Kontextinformationen verwaltet das Betriebssystem:

Benutzer-/Applikationskontext

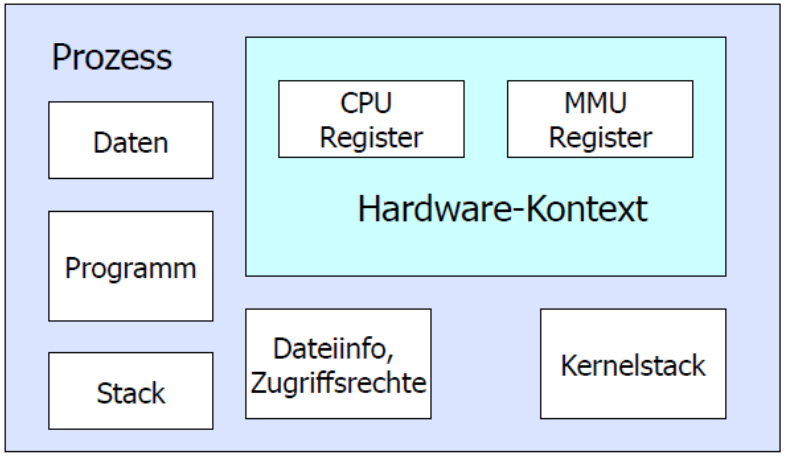
- Daten im zugewiesenen Adressraum (im virtuellen Speicher)

Hardwarekontext

- Register in der CPU

Systemkontext

- Informationen, die das Betriebssystem über einen Prozess speichert



MMU = Memory Management Unit

Die Informationen im Hardwarekontext und Systemkontext verwaltet das Betriebssystem im **Prozesskontrollblock (PCB)**

Hardwarekontext

- Multiprocessing o. -tasking (im Gegensatz zum Multithreading) unterstützt die parallele Ausführung mehrerer Prozesse auf mehreren Prozessoren (CPUs)

- Der **Hardwarekontext** sind die Inhalte der Register in der CPU zum Zeitpunkt der Prozess-Ausführung

- Register, deren Inhalt mindestens bei einem Prozesswechsel gesichert werden müssen:
  - Befehlszähler (*Program Counter*, *Instruction Pointer*) – enthält die Speicheradresse des nächsten auszuführenden Befehls
  - Stackpointer – enthält die aktuelle Stackadresse
  - Basepointer – zeigt auf eine Adresse im Stack
  - Page-table base Register – Adresse, wo die Seitentabelle des laufenden Prozesses anfängt
  - Page-table length Register – Länge der Seitentabelle des laufenden Prozesses
  - Statusregister (Flags...)

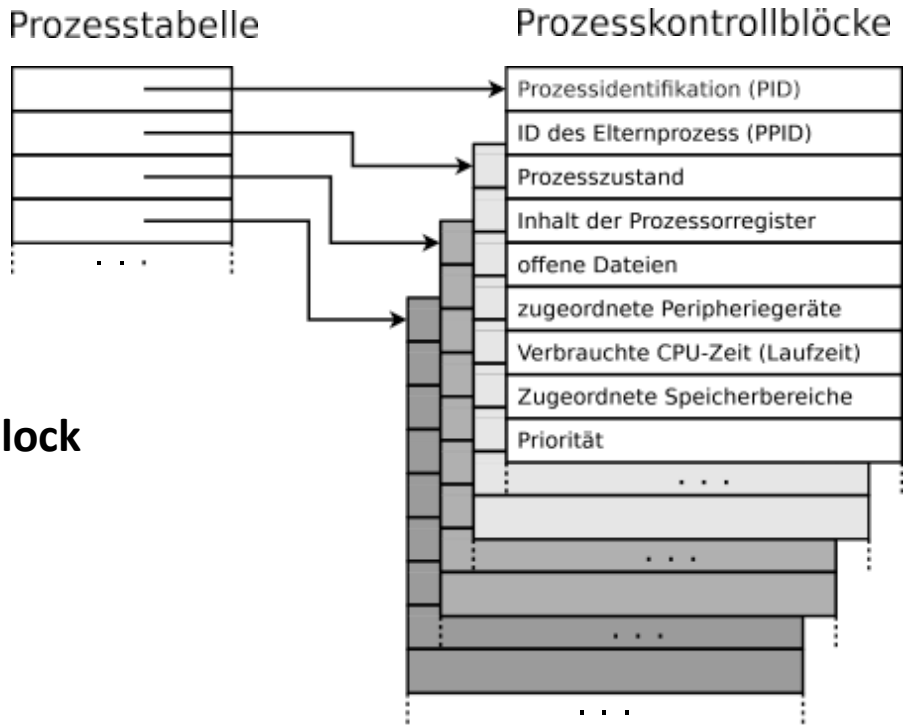
Systemkontext

- Der **Systemkontext** sind die Informationen, die das Betriebssystem über einen Prozess speichert
- Beispiele:
  - Eintrag in der Prozesstabelle
  - Prozessnummer (PID)
  - Prozesszustand
  - Information über Eltern- oder Kindprozesse
  - Priorität
  - Identifier – Zugriffsrechte auf Ressourcen
  - Quotas – Zur Verfügung stehende Menge der einzelnen Ressourcen
  - Laufzeit
  - Geöffnete Dateien
  - Zugeordnete Geräte
  - …

Prozeßtabelle und Prozeßkontrollblöcke

- Jeder Prozess hat seinen eigenen Prozesskontext, der von den Kontexten anderer Prozesse unabhängig ist

- Zur Verwaltung der Prozesse führt das Betriebssystem die **Prozesstabelle**
  - Liste aller existierenden Prozesse
  - enthält für jeden Prozess einen Eintrag, den **Prozesskontrollblock**



Verwaltung von Prozessen und Threads

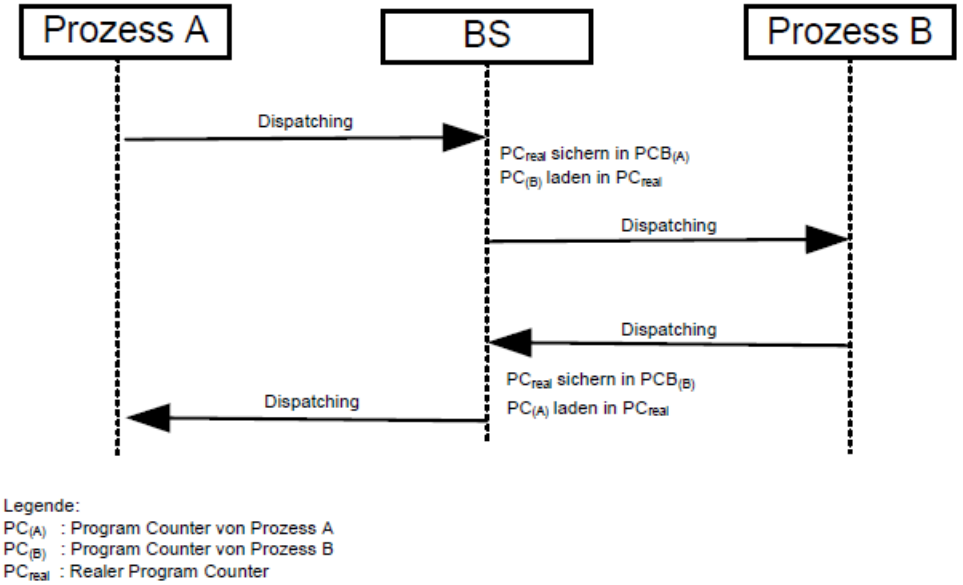
- Process Control Blöcke (PCBs): Datenstruktur jeden Prozesses
- Prozeßtabelle: Enthält alle PCBs
    - Im Kernel, eine für das Gesamtsystem
- Thread Control Blöcke (TCBs) – Einer pro Thread
- Thread Table:
    - Eine pro Prozess in Verwaltung durch die genutzte Bibliothek (bei User-level-Threads=Fibers)
    - Eine für das Gesamtsystem im Kernel (Kernel-(scheduled-)Threads)

Kontextumschaltung (Prozesswechsel)

Beim **Prozess-Scheduling** wechselt der **Scheduler** den aktuell laufenden Thread und (falls nötig) seinen Prozess gegen einen zuvor wartenden Thread eines anderen Prozesses aus:

- Unterbrechung des laufenden Prozesses und Speicherung des Prozeßkontexts in einer Datenstruktur im Speicher
- Restore des Prozeßkontexts des fortzusetzenden Prozesses in die CPU-Register und Fortführen der Ausführung

  - Dauer: ~ ein paar hundert Befehle, 1-2µs



Prozesse und Betriebsmittel

Prozesse **konkurrieren** um die Betriebsmittel (Prozessor, Speicher, Dateien,…)

Beispiel bei nur einer CPU und mehreren Prozessen:

- Prozesse (einer von deren Threads) laufen abwechselnd einige Millisekunden
- Dadurch entsteht der Eindruck paralleler Verarbeitung
- Dazwischen sind neben prozessinternen Threadwechseln auch Prozesswechsel (**Kontextwechsel** oder Kontext-Switch) nötig
    - bisheriger Prozess/ Thread wird gestoppt
    - neuer Prozess/ Thread (re)aktiviert

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Prozess-Scheduling (indirekt durch Thread-Scheduling)

Aufteilung der **CPU-Zeit** auf die berechtigten Threads der Prozesse, basierend auf einem Satz von Optimierungskriterien:

- Durchsatz
- Threadnutzung
- mittlere Antwortzeit
- Energieeffizienz
- stetiger Wechsel gegeben
- Zeitzuteilungs - Fairness
- …

Prozess-/ Thread-Scheduling

Wie würden Sie einen Scheduler schreiben?

… mit Prioritäten?

… mit Zeitscheiben?

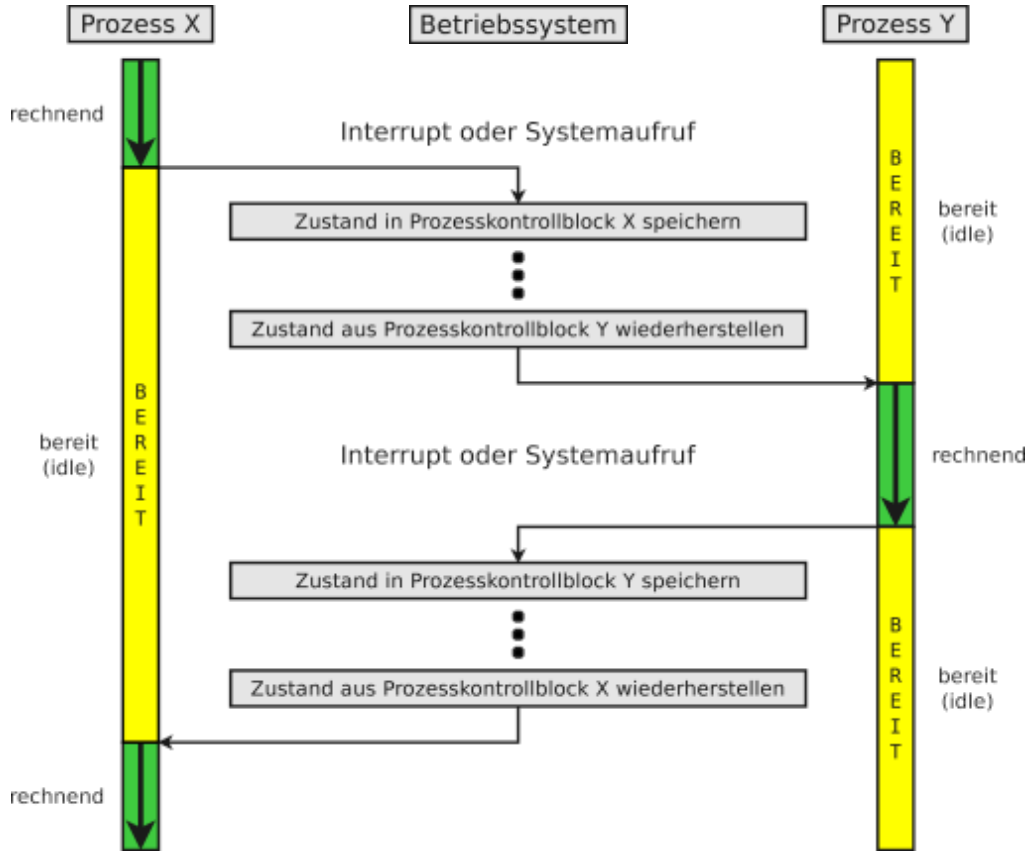
… nach Restlaufzeit?

…

Prozesswechsel

- Beim Prozesswechsel werden der Systemkontext und der Hardwarekontext (⇒ Inhalt der CPU-Register) im PCB gespeichert

- Erhält ein Prozess Zugriff auf die CPU, wird sein Kontext aus dem Inhalt des Prozesskontrollblocks wiederhergestellt



- Jeder Prozess/ jede Task ist zu jedem Zeitpunkt in einem bestimmten **Zustand** ⇒ Zustandsdiagramm der Tasks

Taskzustände

Wir wissen bereits...

Jede Task befindet sich zu jedem Zeitpunkt in einem Zustand

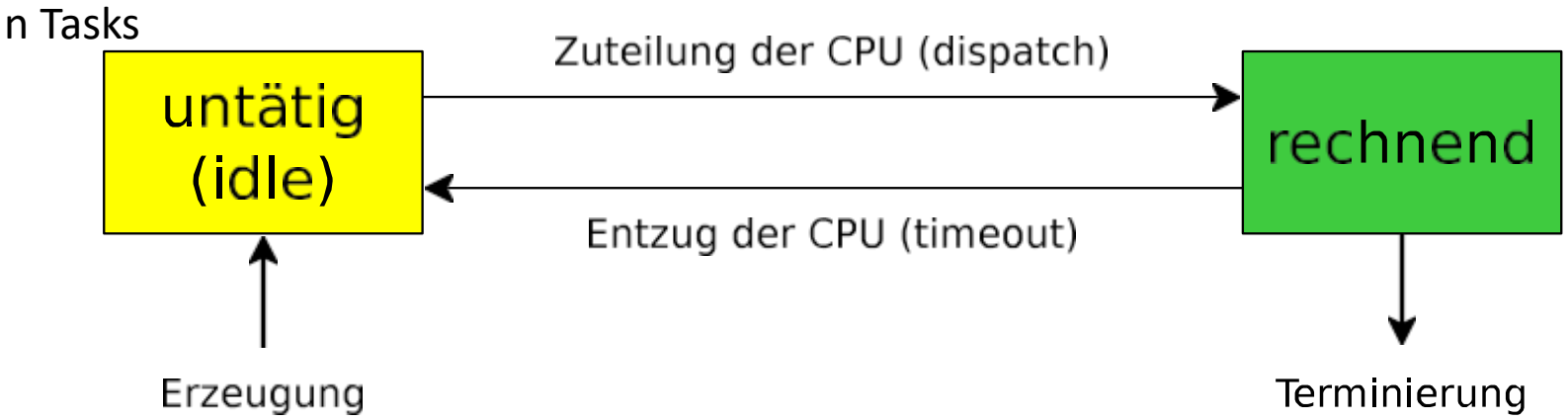
Wie viele unterschiedliche Zustände es gibt,
hängt vom Zustands-Taskmodell des Betriebssystems ab

Wie viele Taskzustände braucht ein Prozessmodell mindestens?

2-Zustands-Taskmodell

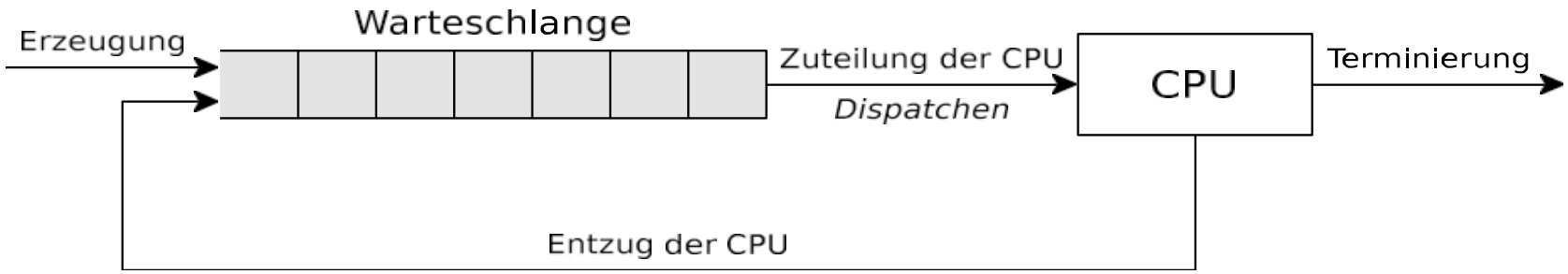
Prinzipiell genügen 2 Taskzustände:

rechnend: Einem Task wurde die CPU zugeteilt
untätig (*idle*): Task wartet auf die Zuteilung der CPU



2-Zustands-Taskmodell (Implementierung)

- Die Tasks im Zustand **idle** müssen in einer Warteschlange gespeichert werden, in der sie auf ihre Ausführung warten
  - Die Liste wird nach Taskpriorität oder Wartezeit sortiert



Die Priorität hat unter Linux einen Wert von -20 bis +19 (in ganzzahligen Schritten).
-20 ist die höchste Priorität und 19 die niedrigste Priorität. Die Standardpriorität ist 0.
Normale Nutzer können Prioritäten von 0 bis 19 vergeben. Der Systemverwalter (root) darf auch negative Werte vergeben.

- Dieses Modell zeigt auch die Arbeitsweise des **Dispatchers**
  - Aufgabe des Dispatchers ist die **Umsetzung** der Zustandsübergänge
- Die **Ausführungsreihenfolge** der Prozesse legt der **Scheduler** fest, der einen **Scheduling-Algorithmus** verwendet

Konzeptioneller Fehler des 2-Zustands-Taskmodells

Das 2-Zustands-Taskmodell geht davon aus, dass alle Tasks immer zur Ausführung bereit sind

- Das ist unrealistisch!

Es gibt fast immer Tasks, die **blockiert** sind, u.a. aus folgenden Gründen:

- Warten auf die Eingabe oder Ausgabe eines E/A-Geräts
- Warten auf das Ergebnis einer anderen Task
- Warten auf eine Reaktion des Benutzers

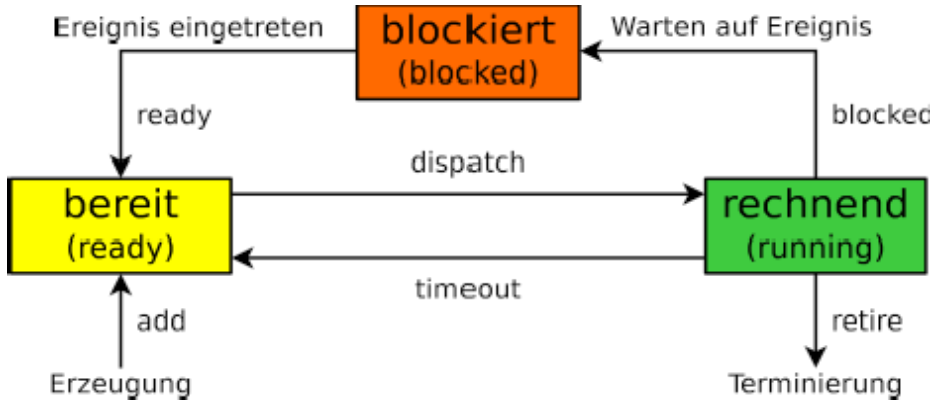
Lösung: Die untätigen Tasks werden in 2 Gruppen unterschieden

- Tasks, die **bereit** (ready) sind
- Tasks, die **blockiert** (blocked) sind

⇒ 3-Zustands-Taskmodell

3-Zustands-Taskmodell

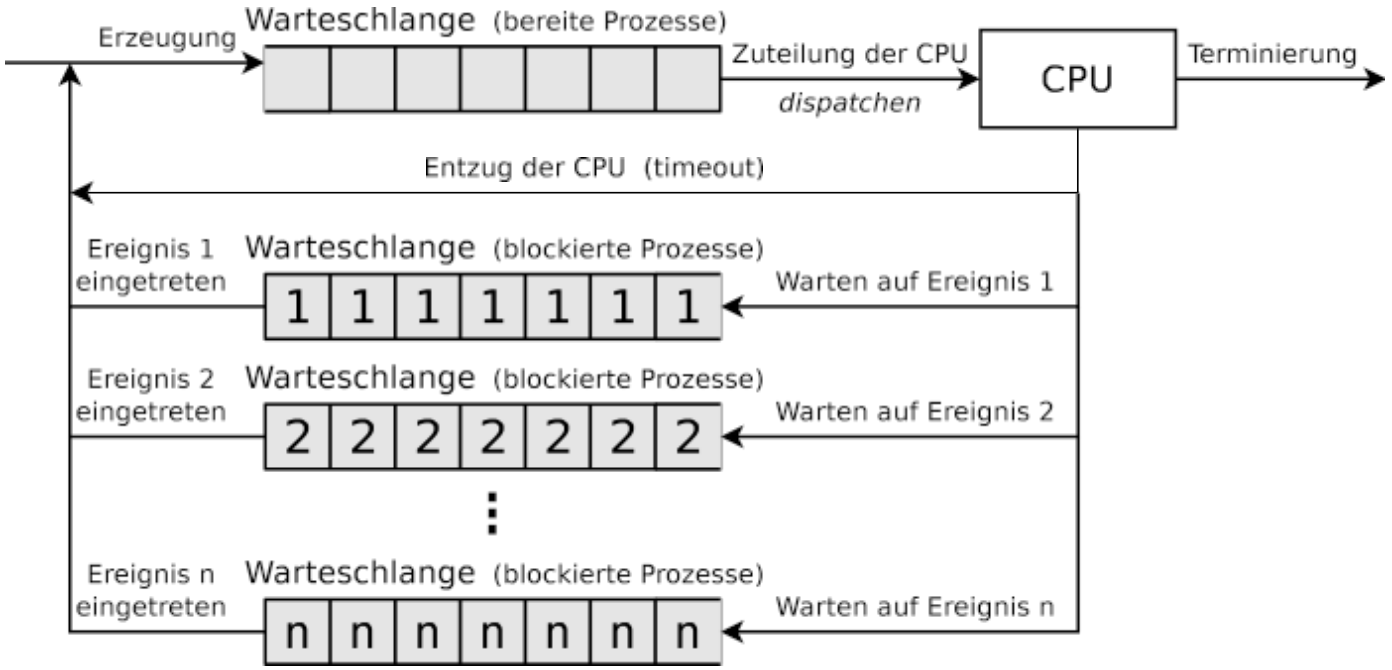
Jede Task befindet sich in einem der folgenden Zustände:



- **rechnend** (*running*):
  - Die Task hat Zugriff auf die CPU und führt auf dieser Instruktionen aus
- **bereit** (*ready*):
  - Die Task könnte unmittelbar Instruktionen auf der CPU ausführen und wartet aktuell auf die Zuteilung der CPU
- **blockiert** (*blocked*):
  - Die Task kann momentan nicht weiter ausgeführt werden und wartet auf das Eintreten eines Ereignisses oder die Erfüllung einer Bedingung
  - Dabei kann es sich z.B. um eine Nachricht einer anderen Task oder eines Eingabe-/Ausgabegeräts oder um das Eintreten eines Synchronisations-ereignisses handeln

3-Zustands-Taskmodell (Realisierung)

- In der Praxis implementieren Betriebssysteme (z.B. Linux) mehrere Warteschlangen für Tasks im Zustand **blockiert**

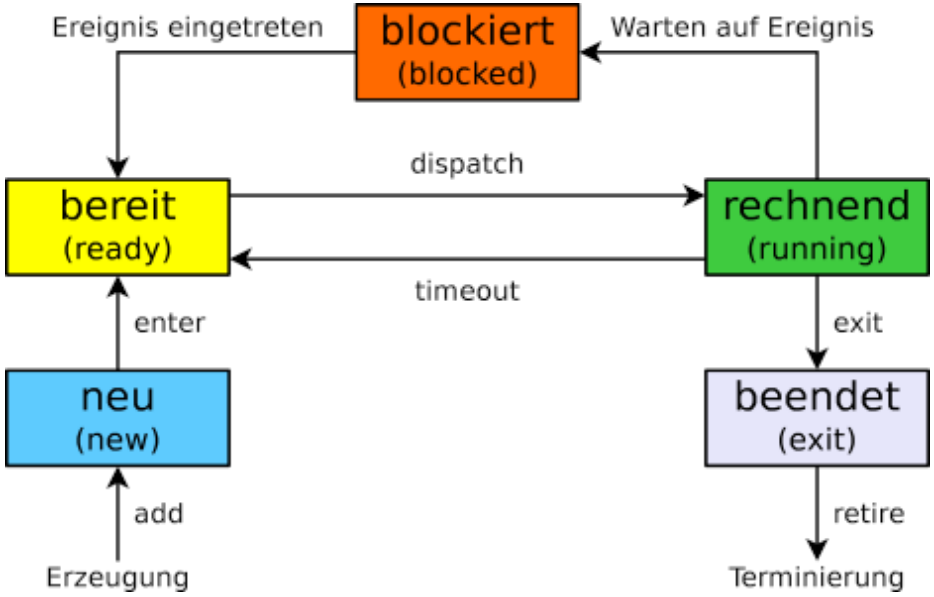


- Beim Zustandsübergang wird der Thread-/ Prozesskontrollblock der Task aus der alten Zustandsliste entfernt und in die neue Zustandsliste eingefügt

- Für Tasks im Zustand **rechnend** existiert keine eigene Liste (1 pro Core)

5-Zustands-Taskmodell

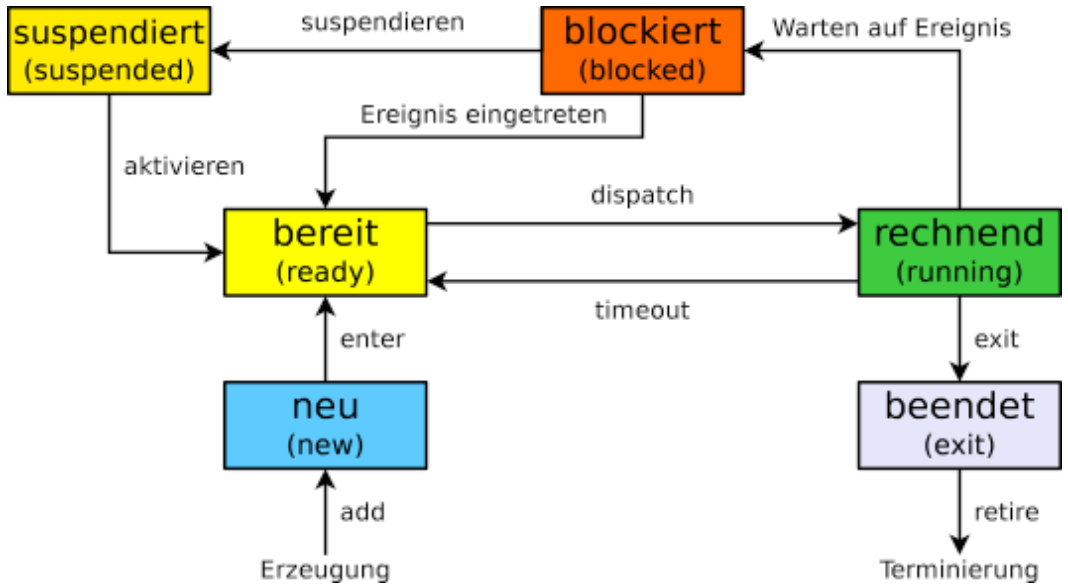
- Es ist empfehlenswert, das 3-Zustands-Taskmodell um 2 weitere Taskzustände zu erweitern:
  - **neu** (*new*): Die Task (Prozess-/Threadkontrollblock) ist erzeugt, wurde aber vom Betriebssystem noch nicht in die Warteschlange für Tasks im Zustand **bereit** eingefügt
  - **exit**: Die Task ist fertig abgearbeitet oder wurde beendet, aber ihr Thread-/ Prozesskontrollblock existiert aus verschiedenen Gründen noch
- Grund für die Existenz der Taskzustände **neu** und **exit**:
  - Auf manchen Systemen ist die Anzahl der ausführbaren Tasks limitiert, um Speicher zu sparen und den Grad des Mehrprogramm-betriebs festzulegen



6-Zustands-Taskmodell

- Ist nicht genügend physischer Hauptspeicher für alle Prozesse verfügbar, müssen Teile von Prozessen ausgelagert werden ⇒ **Swapping**
- Das Betriebssystem lagert Taskkontexte aus, die im Zustand **blockiert** sind (-> suspendiert)
- Dadurch steht den Tasks in den Zuständen **rechnend** und **bereit** mehr Hauptspeicher zur Verfügung

  - Es macht also Sinn, das 5-Zustands-Taskmodell um den Taskzustand **suspendiert** (*suspended*) zu erweitern



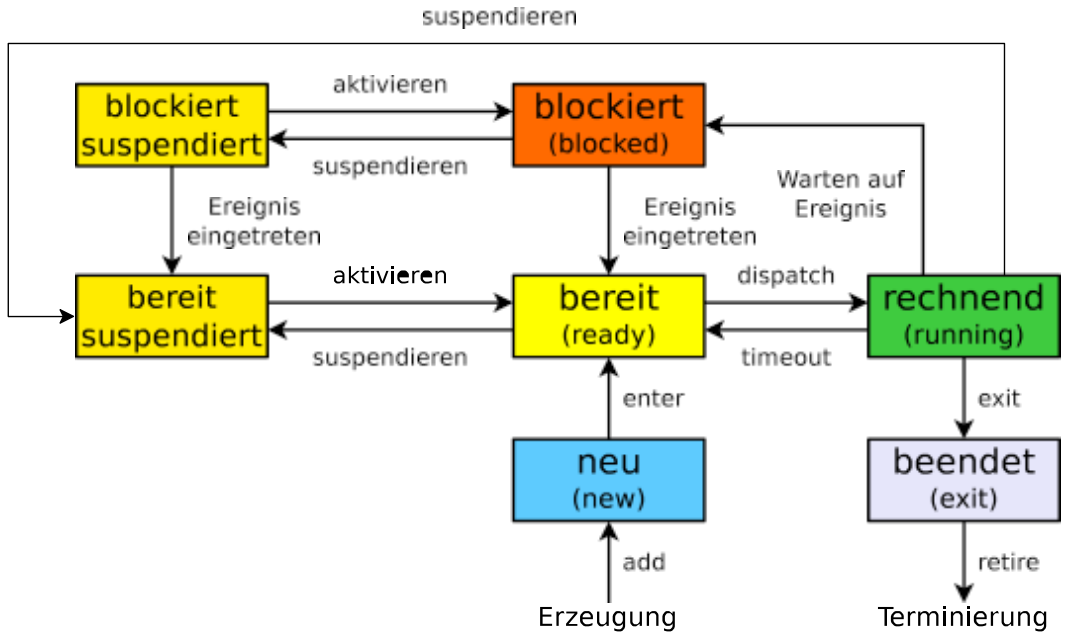
7-Zustands-Taskmodell

- Wurde ein Prozess suspendiert, ist es besser, den frei gewordenen Platz im Hauptspeicher zu verwenden, um einen anderen ausgelagerten Prozess zu aktivieren, als ihn einem neuen Prozess zuzuweisen.
  Das ist aber nur dann sinnvoll, wenn der suspendierte Prozess nicht (mehr) blockiert ist

- Im 6-Zustands-Taskmodell fehlt die Möglichkeit, die ausgelagerten (suspendierten) Prozesse zu unterscheiden in:

  - blockierte ausgelagerte Prozesse

  - nicht-blockierte ausgelagerte Prozesse



Task-Lebenszyklus

Eine Task wird mit Mitteln des Betriebssystems erzeugt:

- Realen Prozessor, Hauptspeicher und weitere Ressourcen zuordnen
- Programmcode und Daten in Speicher laden
- Gegebenenfalls Prozesskontext laden und Prozess starten

Für das Beenden einer Task gibt es mehrere Gründe:

- Normaler Exit
- Error exit (vom Programmierer gewünscht/ verursacht, fatal error)
- Durch eine andere Task beendet (killed)

Prozeß-Erzeugung

- der UNIX-Ansatz: **fork** & **exec**
  - **fork()** erzeugt eine identische Kopie des aktuellen Prozesses, inkl. Ausführungskontext
  - der neue Prozess **erbt** die Eigenschaften des Vorfahren (außer der PID und PPID)
  - **exec*()** setzt für das laufende Executable einen anderen Programmkontext (z.B. im Kind, wenn Returnwert von fork() == 0)

- der Windows-Ansatz: **CreateProcess()**
  - Prozesse werden direkt durch einen anderen Prozess erzeugt (unter Angabe des Ausführungkontexts und Programms)
  - Jeder Prozess erhält zur Verwaltung ein Objekt-Handle mit **PID** (Idle-Prozess hat PID 0)
  - ein neuer Prozess erbt nichts automatisch

Isolierung

- Prozesse sind naturgemäß gegeneinander **isoliert,** um Daten-Vertraulichkeit zu gestatten und sich gegen bösartige Prozesse zu schützen
  - Hardwareunterstützt für höhere Effizienz
  - Durch zwei Prozesse zugreifbarer Speicher muss explizit geteilt (gemeinsam genutzt) werden
  - unterstützt durch **Virtual memory abstraction**

Virtueller Speicher

Segmentierung

- Prozesse können abgegrenzte Segmente des Hauptspeichers nutzen und adressieren über **Offset- und Längen-Register**

Paging

- Der Systemspeicher erscheint allen Prozessen als kontinuierliches Array
- Willkürlicher Zugriff erzeugt „Access violations" (**page faults**)
- Das System pflegt **Page tables,** um die Zugriffsarten auf die Speicherregionen festzulegen (read / write / execute)

Speichermanagement

Paging darf zuviel Speicher zusagen (Details unter „Speichermanagement“):

- jeder Prozeß kann mehr Speicher **reservieren**, als physikalisches RAM verfügbar ist
- um Speicher **bereitzustellen**, muß das System die virtuellen Speicherseiten auf physikalische Seiten mappen (**frame**)
- sind keine leeren Frames verfügbar, erfolgt eine Seitenersetzung:
  - UNIX: Page wird in die “Swap partition” ausgelagert (irreführender Name!)
  - Windows: schreibt Seiteninhalt in das Pagefile

Kommunikation

Prozesse sind gegeneinander isoliert, **Interprozeß-Kommunikation** muß explizit stattfinden, z.B. durch:

- Pipes
- Files
- Shared memory
- Sockets

Schutz

Prozesse sind nicht nur voneinander isoliert, **sondern auch vom Betriebssystem**:

- können auf Kernel-Datenstrukturen nicht zugreifen
- können Kernel-Funktionen nicht explizit aufrufen

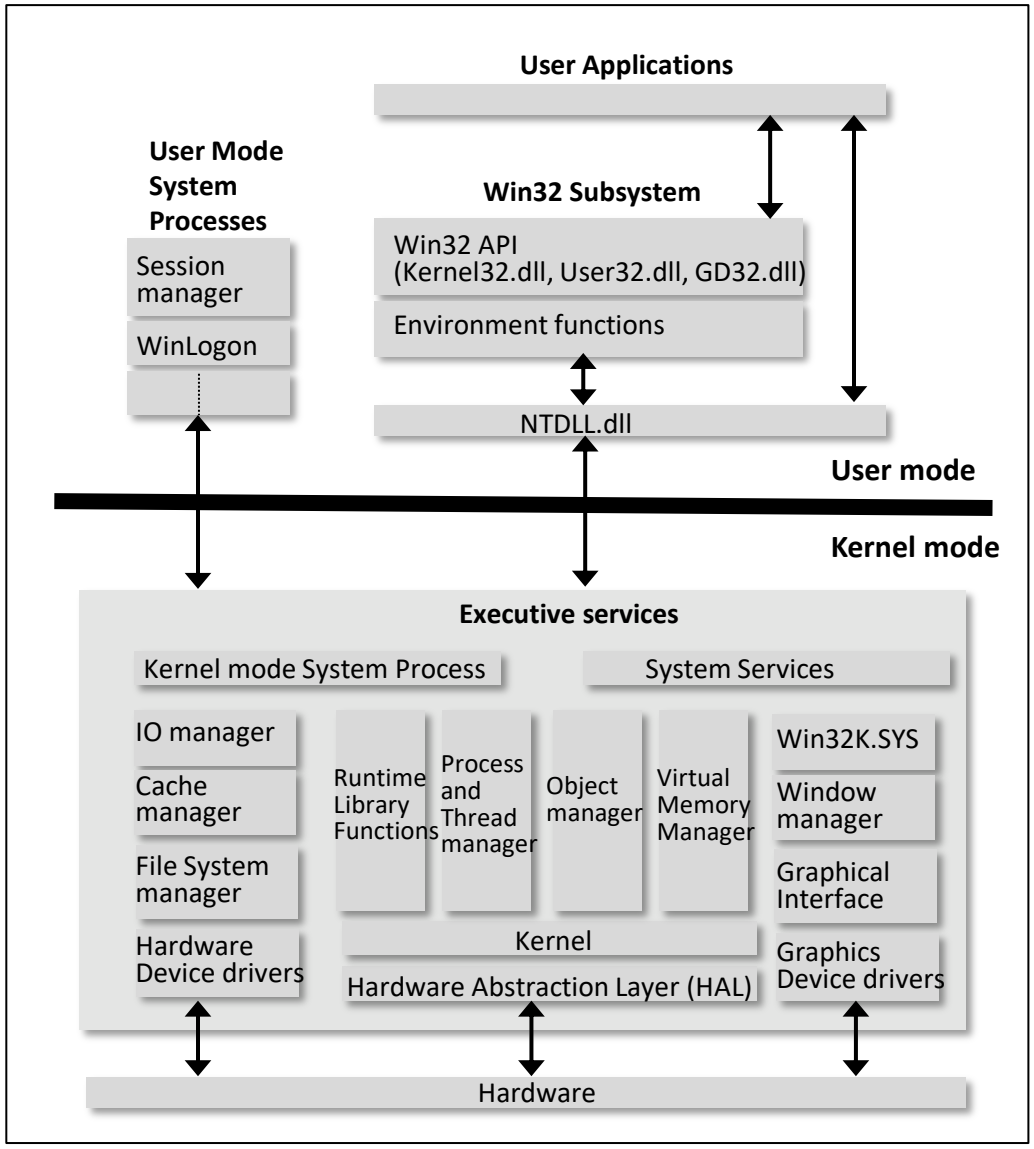
- Die Systemverarbeitung geschieht separiert in **User-Mode** und **Kernel- / privilegierten Mode** (Hardware- Support)
- **System calls** machen Kernel –Funktionalität zugänglich für User mode - Prozesse

System calls

= Calls vom User mode in den Kernel mode durch **Software interrupt** (**trap**)

- *System call* betritt den Kernel mode durch eine **Interrupt service routine** (ISR)
  - "Single entry point" in den Kernel
    (Windows: System call hat User- und Kernel-mode-Anteile)
- Kernel validiert Argumente und stellt den Funktionsaufruf durch
- Thread-Ausführung wird suspendiert, bis der *System call* zurückkehrt
- *System calls* werden nicht direkt aufrufen, sondern über Wrapper Funktionen

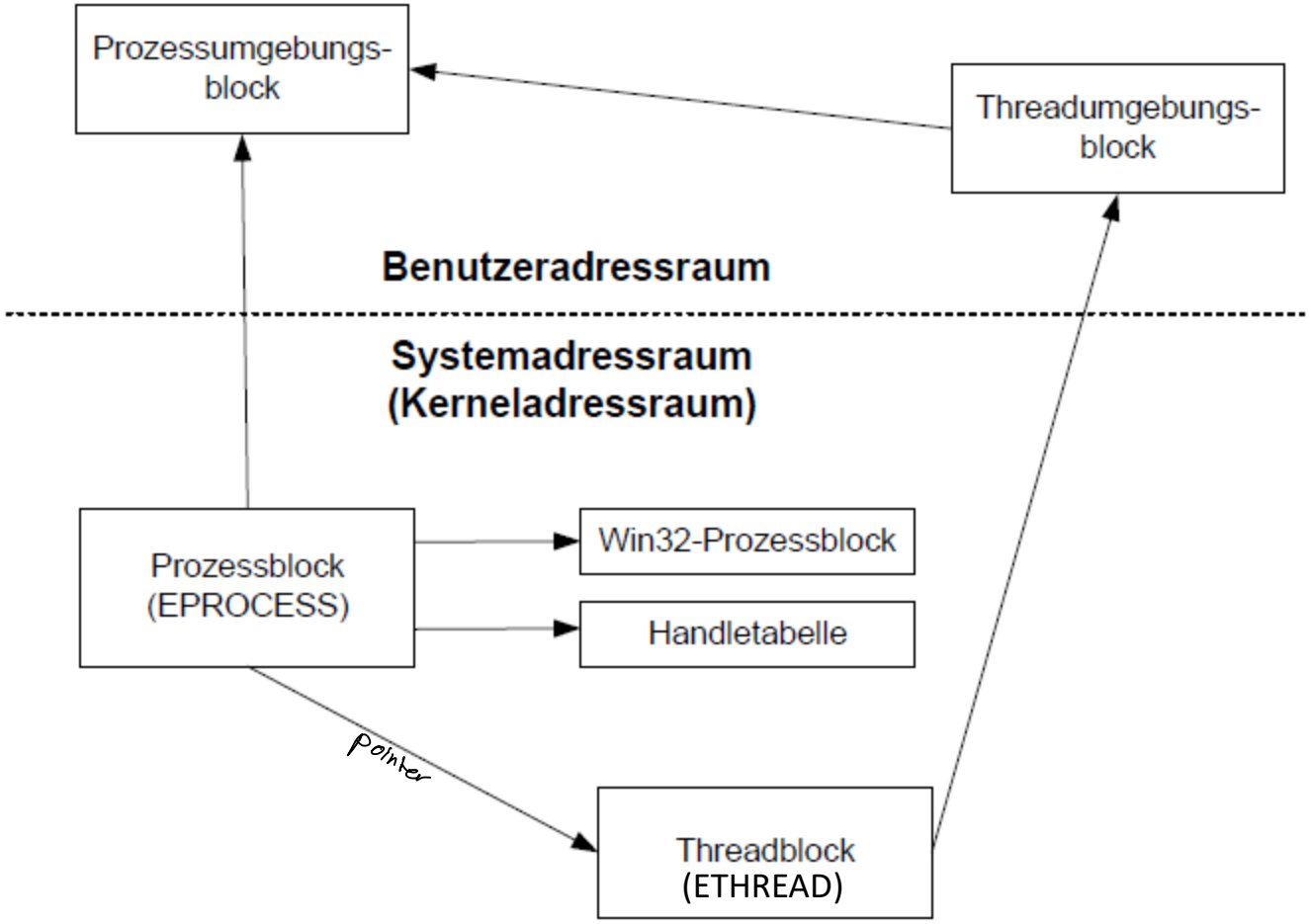
System calls (Windows)



Input/ Output

- I/O-System-Interfaces zwischen Threads und **Geräten**
  - **Libraries**, **System calls** und **Driver** mappen einen kleinen Satz von Standard-funktionen auf komplexe Gerätespezifika
  - **Hardwareabstraktion**
- **Polling** / **Interrupts** / **DMA**
- Threads können auf I/O- Completion **warten**
  - relevant für Scheduling

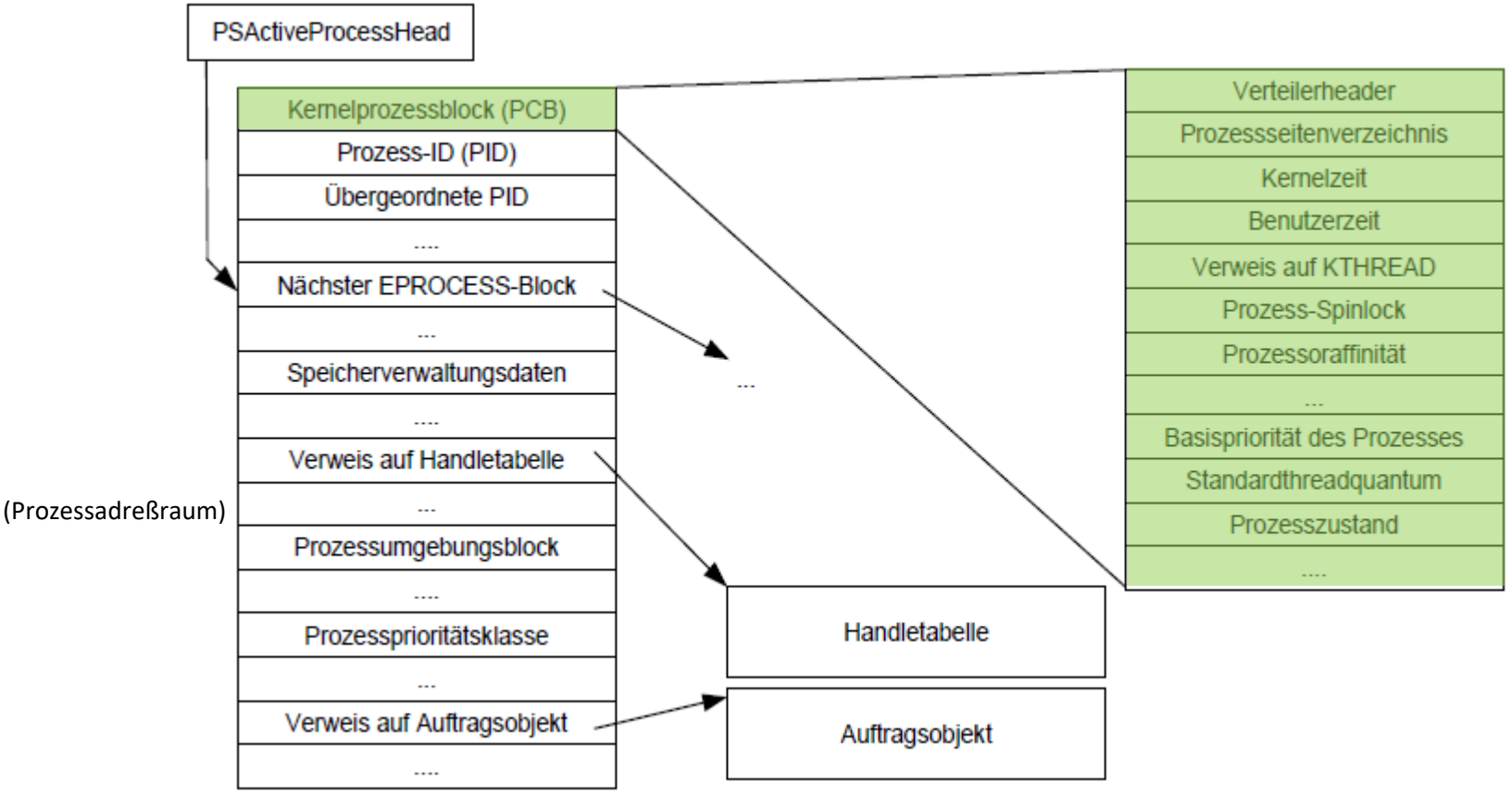
Prozessmodell unter Windows



Quelle: *Solomon, D. A.; Russinovich, M.:* Microsoft Windows Internals, Microsoft Press, Part 1 und 2, 6. Auflage, 2013

Prozessmodell unter Windows

Der EPROCESS- (Executive process) Block enthält wichtige Informationen zum Prozess :



Quelle: *Solomon, D. A.; Russinovich, M.:* Microsoft Windows Internals, Microsoft Press, Part 1 und 2, 6. Auflage, 2013

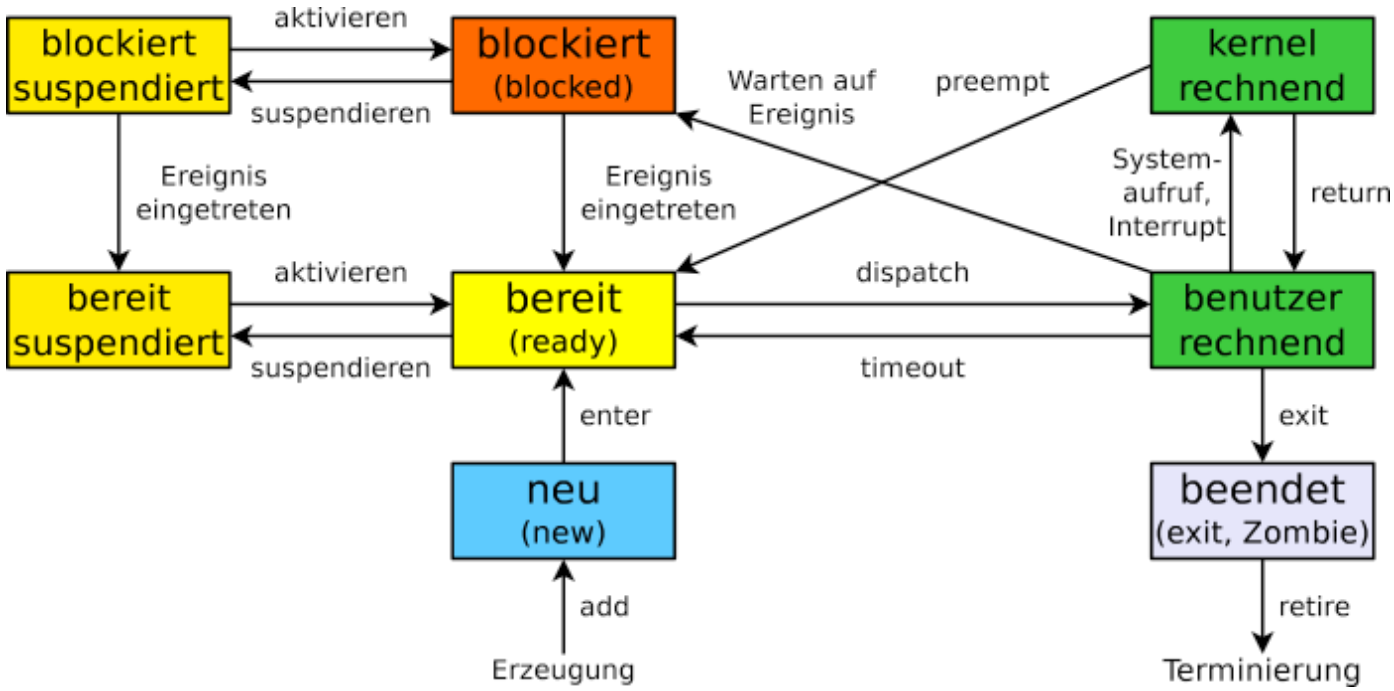
Shell - Prozeß

- Ein Prozess, der:
  - erzeugt wird beim **User-Login**
  - zugehörig ist zum angemeldeten Benutzer
  - **Benutzereingaben** vom **Terminal** erhält und interpretiert
- Shell-Prozeß ist in der Lage, neue **Prozesse zu erzeugen** durch eigene Duplizierung (UNIX) oder explizites Erzeugen eines Prozesses durch ein Programm (Windows)
- interaktives **Interface zum Betriebssystem** und seinen Features (Redirections, Pipes, Job control, Scripting, ...)

Taskmodell von Linux/UNIX (etwas vereinfacht)

Der Zustand **rechnend** (*running*) wird unterteilt in die Zustände...

- **Benutzer rechnend** (*user running*) für Tasks im Benutzermodus
- **Kernel rechnend** (*kernel running*) für Tasks im Kernelmodus



Ein **Zombie-Prozess** ist fertig abgearbeitet (via Systemaufruf `exit`), aber sein Eintrag in der Prozesstabelle existiert so lange, bis der Elternprozess den Rückgabewert (via Systemaufruf `wait`) abgefragt hat.

Prozesse unter Linux/UNIX erzeugen mit fork (1/2)

- Der Systemaufruf *fork()* ist die üblicherweise verwendete Möglichkeit, unter UNIX einen neuen Prozess durch Kopie zu erzeugen und sofort zu starten
- Ruft ein Prozess *fork()* auf, wird eine identische Kopie als neuer Prozess erzeugt
  - Der aufrufende Prozess heißt **Vaterprozess** oder **Elternprozess**
  - Der neue Prozess heißt **Kindprozess**
- Der Kindprozess läuft nach der Erzeugung über den gleichen Programmcode (kehrt zusätzlich zum Vater aus der Funktion *fork()* zurück!)
- Auch die Befehlszähler haben den gleichen Wert, verweisen also auf die gleiche Zeile im Programmcode
- Geöffnete Dateien und Speicherbereiche des Elternprozesses werden für den Kindprozess kopiert und sind unabhängig vom Elternprozess (Stackinhalt inkl. lokaler Variabler wird kopiert)
  - Kindprozess und Elternprozess besitzen ihren eigenen Prozesskontext (inkl. PID)

Mit `vfork` existiert eine Variante von `fork`, die nicht den Adressraum des Elternprozesses kopiert, und somit weniger Verwaltungsaufwand als `fork` verursacht. Die Verwendung von `vfork` ist sinnvoll, wenn der Kindprozess direkt nach seiner Erzeugung durch einen anderen Kontext ersetzt werden soll.

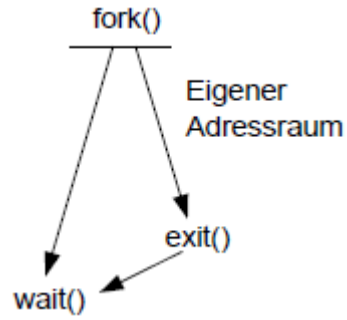
Prozesse unter Linux/UNIX erzeugen mit fork (2/2)

- Ruft ein Prozess *fork()* auf, wird eine exakte Kopie erzeugt
  - Die Prozesse (Original und Kopie) unterscheiden sich nur in den Rückgabewerten bei der Rückkehr aus *fork()* (kehrt in 2 Prozesse zurück!)

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4
5  void main() {
6    int rueckgabewert = fork();
7
8    if (rueckgabewert < 0) {
9      // Hat fork() den Rückgabewert -1, ist ein Fehler aufgetreten.
10     // Speicher oder Prozesstabelle sind voll.
11     ...
12   }
13   if (rueckgabewert > 0) {
14     // Hat fork() einen positiven Rückgabewert, sind wir im Elternprozess.
15     // Der Rückgabewert ist die PID des neu erzeugten Kindprozesses.
16     ...
17   }
18   if (rueckgabewert == 0) {
19     // Hat fork() den Rückgabewert 0, sind wir im Kindprozess.
20     ...
21   }
22 }
```

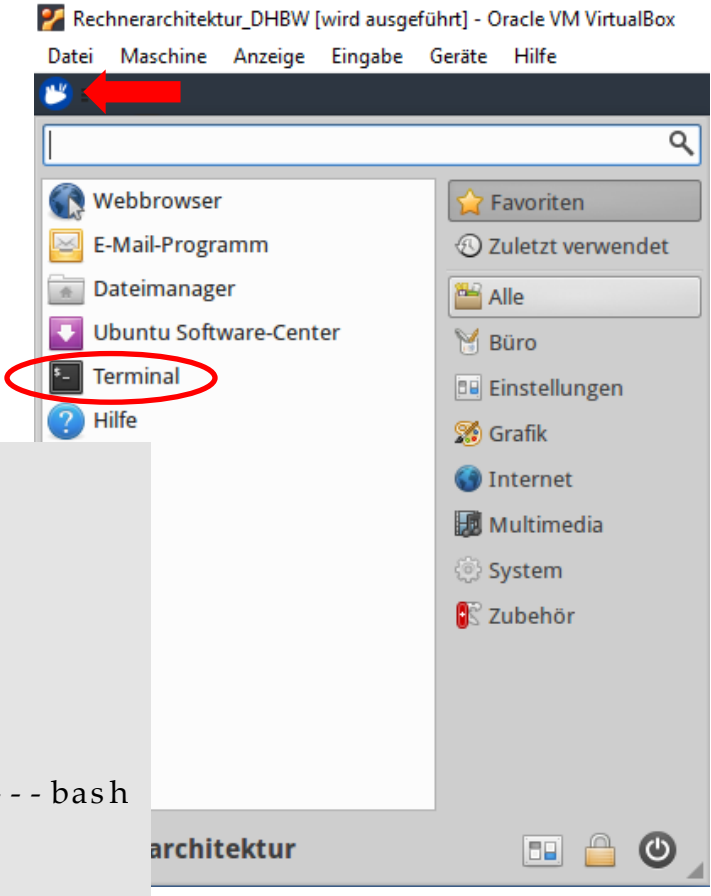
Prozesse unter Linux/UNIX erzeugen mit fork (2/2)

```
void main()
{
    int ret;                // Returncode von fork
    int status;             // Status des Kindprozesses
    pid_t pid;              // pid_t ist ein spezieller Datentyp, der eine PID beschreibt
    ret = fork();           // Kindprozesses wird zusätzlich erzeugt
    if (ret == 0)
    {                       // Anweisungen, die im Kindprozess ausgeführt werden sollen
        ...
        exit(0);            // Beenden des Kindprozesses mit Status 0 (ok)
    }
    else
    {                       // Anweisungen, die parallel nur im Elternprozess ausgeführt werden sollen
                            // Zur Ablaufzeit kommt hier nur der Elternprozess hin (Returncode = PID > 0)
        ...
        pid = wait(&status);    // Optionales Warten auf das Ende des Kindprozesses
        exit(0);                // Beenden des Vaterprozesses mit Status 0 (ok)
    }
}
```



Prozeßbaum

- Durch das Erzeugen immer neuer Kindprozesse mit *fork()* entsteht ein beliebig tiefer Baum von Prozessen (⇒ Prozesshierarchie)
- Das Kommando pstree gibt die laufenden Prozesse unter Linux/UNIX als Baum entsprechend ihrer Eltern-/Kind-Beziehungen aus
- (ausprobieren in der Virtual Box (aus dem Fach Betriebssysteme unter xubuntu))



initial Prozess id = 1

```
$ pstree
init-+-Xprt
     |-acpid
     ...
     |-gnome-terminal-+-4*[bash]
     |                |-bash---su---bash
     |                |-bash-+-gv---gs
     |                |      |-pstree
     |                |      |-xterm---bash---xterm---bash
     |                |      |-xterm---bash---xterm---bash---xterm---bash
     |                |      `-xterm---bash
     |                |-gnome-pty-helpe
     |                `-{gnome-terminal}
     |-4*[gv---gs]
```

Informationen über Prozesse unter Linux/UNIX

```
$ ps -eFw
UID    PID  PPID  C    SZ   RSS PSR STIME TTY          TIME CMD
root     1     0  0 51286  7432   2 Apr11 ?      00:00:03 /sbin/init
root  1073     1  0 90930  6508   0 Apr11 ?      00:00:00 /usr/sbin/lightdm
root  1551  1073  0 60913  6772   2 Apr11 ?      00:00:00 lightdm --session-child 14 23
bnc   2143  1551  0  1069  1560   0 Apr11 ?      00:00:00 /bin/sh /etc/xdg/xfce4/xinitrc
bnc   2235  2143  0 85195 18888   3 Apr11 ?      00:00:11 xfce4-session
bnc   2284  2235  0 110875 45256  3 Apr11 ?      00:06:20 xfce4-panel --display :0.0
bnc   2389  2235  0 129173 47904  0 Apr11 ?      00:00:26 xfce4-terminal --geometry=80x24
bnc   2471  2389  0  5374  5360   2 Apr11 pts/0  00:00:00 bash
bnc   2487     1  5 316370 395892 0 Apr14 ?      00:08:58 /opt/google/chrome/chrome
bnc   2525  2389  0  5895  6620   3 Apr11 pts/5  00:00:00 bash
bnc   3105  2284  0 597319 257520 0 Apr11 ?      00:05:22 kate -b
bnc   3122  3105  0  5364  5156   2 Apr11 pts/6  00:00:00 /bin/bash
bnc  11196  2471  0 269491 181048 0 Apr14 pts/0  00:00:25 okular bsrn_vorlesung_04.pdf
bnc  16325     1  0 346638 146872 3 10:31 ?      00:00:16 evince BA.pdf
bnc  17384  2525  1 223478  61312  2 10:39 pts/5  00:00:49 dia
bnc  19561  2471  0  9576  3340   0 11:20 pts/0  00:00:00 ps -eFw
```

C (CPU) = CPU-Belastung des Prozesses in Prozent
SZ (Size) = Virtuelle Prozessgröße = Textsegment, Heap und Stack
RSS (Resident Set Size) = Belegter physischer Speicher (ohne Swap) in kB
PSR = Dem Prozess zugewiesener CPU-Kern
STIME = Startzeitpunkt des Prozesses
TTY (Teletypewriter) = Steuerterminal. Meist ein virtuelles Gerät: pts (pseudo terminal slave)
TIME = Bisherige Rechenzeit des Prozesses auf der CPU (HH:MM:SS)

Unabhängigkeit von Eltern- und Kindprozeß

- Dieses Beispiel zeigt, dass Eltern- und Kindprozess unabhängig voneinander arbeiten und unterschiedliche Speicherbereiche verwenden

```
1  # include  <stdio .h>
2  # include  <unistd .h> #
3  include  <stdlib .h>
4
5  void  main ()  {
6     int i;
7        if  ( fork ())
8           // Hier arbeitet der  Vaterprozess
9           for  ( i  =  0;  i  <  5000000;  i++)
10             printf("\n  Vater:  %i",  i);
11       else
12          // Hier arbeitet der  Kindprozess
13          for  ( i  =  0;  i  <  5000000;  i++)
14             printf("\n  Kind  :  %i",  i);
15 }
```

```
Kind  :  0
Kind  :  1
. . .
Kind  :  21019
Vater:  0
. . .
Vater:  50148
Kind  :  21020
. . .
Kind  :  129645
Vater:  50149
. . .
Vater:  855006
Kind  :  129646
. . .
```

- In der Ausgabe sind die Prozesswechsel zu sehen
- Der Wert der Schleifenvariablen i beweist, dass Eltern- und Kindprozess unabhängig voneinander sind
  - Das Ergebnis der Ausführung ist nicht reproduzierbar

Die PID-Nummern von Eltern- und Kindprozess (1/2)

```
1  #include  <stdio .h>
2  #include  <unistd .h>
3  #include  <stdlib .h>
4
5  void  main ()
6  {
7     int  pid_des_Kindes ;
8     pid_des_Kindes  =  fork ();  // potentielle Prozessumschaltung
9
10    // Es  kam zu  einem  Fehler  --> Programmabbruch
11    if ( pid_des_Kindes < 0)
12    {  perror("\n Es  kam  bei  fork () zu  einem  Fehler!");
13       exit (1);
14    }
15
16    // Vaterprozess
17    if  ( pid_des_Kindes  >  0)
18    {  printf("\n Vater: PID : %   i", getpid ());  printf("\n  Vater:  PPID :  %i  ",
19                  getppid ());
20    }
21
22    // Kindprozess
23    if  ( pid_des_Kindes  ==  0)
24    {  printf("\n  Kind :    PID :  %i",  getpid ());  printf("\n  Kind :    PPID :  %i",
25       getppid ());
26    }
27 }
```

- Das Beispiel erzeugt einen Kindprozess
- Kindprozess und Elternprozess (parent) geben beide aus:
  - Eigene PID
  - PID des Vaters (PPID)

Die PID-Nummern von Eltern- und Kindprozess (2/2)

Die Ausgabe ist üblicherweise mit dieser vergleichbar:

```
Vater:  PID :   20952
Vater:  PPID :  3904
Kind:   PID :   20953
Kind:   PPID :  20952
```

Gelegentlich kann man folgendes Ereignis beobachten:

```
Vater:  PID :   20954
Vater:  PPID :  3904
Kind:   PID :   20955
Kind:   PPID :  1
```

Der Elternprozess wurde vor dem Kind-Prozess beendet

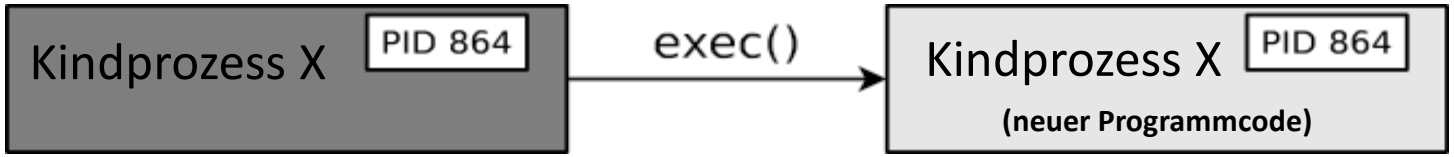
- Wird der Elternprozess vor dem Kindprozess beendet, bekommt er *init* als neuen Elternprozess zugeordnet
- Elternlose Prozesse werden immer von *init* adoptiert

init (PID 1) ist der erste/ initiale Prozess unter Linux/UNIX
Alle laufenden Prozesse stammen von `init` ab ⇒ `init` = Vater aller Prozesse

Prozesskontext ersetzen mit *exec* ()*

- Der Systemaufruf exec*() ersetzt einen Prozesskontext durch einen anderen
  - Es findet eine **Verkettung** statt
  - Der veränderte Prozess behält die PID
- Will man aus einem Prozess heraus ein Programm starten, ist es nötig, zuerst mit fork() einen neuen Prozess zu erzeugen und dann dessen Kontext mit exec{ve}() zu ersetzen (Prozess kann Kontext nur selbst ersetzen)

- Schritte einer Programmausführung in der Shell:
  - Die Shell erzeugt mit fork() eine identische Kopie von sich selbst
  - Im neuen Prozess wird mit execve() das eigentliche Programm gestartet



Beispiel zum Systemaufruf *exec ()*

```
$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
user        1772    1727  0 May18 pts/2    00:00:00 bash
user       12750    1772  0 11:26 pts/2    00:00:00 ps -f
$ bash
$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
user        1772    1727  0 May18 pts/2    00:00:00 bash
user       12751    1772 12 11:26 pts/2    00:00:00 bash
user       12769   12751  0 11:26 pts/2    00:00:00 ps -f
$ exec ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
user        1772    1727  0 May18 pts/2    00:00:00 bash
user       12751    1772  4 11:26 pts/2    00:00:00 ps -f
$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
user        1772    1727  0 May18 pts/2    00:00:00 bash
user       12770    1772  0 11:27 pts/2    00:00:00 ps -f
```

andere bash beendet

- Durch das exec hat ps - f den Kontext der Bash ersetzt und deren PID (12751) und PPID (1772) übernommen

Ein weiteres Beispiel zu *exec ()*

```
1  #include  <stdio .h>
2  #include  <unistd .h>
3
4  int  main ()  {
5          int  pid ;
6          pid  =  fork ();
7
8          //  Wenn die PID  !=0  -->Elternprozess
9          if ( pid ) {
10                 printf("... Elternprozess ...\n");
11                 printf("[ Eltern ]  Eigene PID : %d \n",
12                 getpid ());  printf("[ Eltern ]  PID des Kindes:  %d\n",  pid );
13         }
14         //  Wenn die PID =0  -->  Kindprozess
15         else {
16             printf("... Kindprozess ...\ n");
17             printf("[ Kind ]  Eigene  PID : %d\n",  getpid ());
18             printf("[ Kind ]  PID  des Vaters:  %d\n",  getppid ());
19
20             // Aktuelles Programm durch "date" ersetzen
21             // "date" wird der Prozessname in der  Prozesstabelle
22             execl("/ bin / date ",  " date ",  "-u",  NULL );
23       }  //  fuehrt „date –u“ aus – Ausgabe Datum in UTC
24       printf("[%d ] Programmende \n", getpid ());
25       return 0;
26 }
```

- Der Systemruf exec() existiert nicht als Bibliotheksfunktion
- Aber es existieren mehrere Varianten der Funktion exec()
- Eine Variante ist execl() mit den Argumenten: Pfad und weiteren…

Hilfreiche Übersicht über die verschiedenen Varianten der Funktion `exec()`
https://man7.org/linux/man-pages/man3/exec.3.html

Erklärung zum exec() - Beispiel

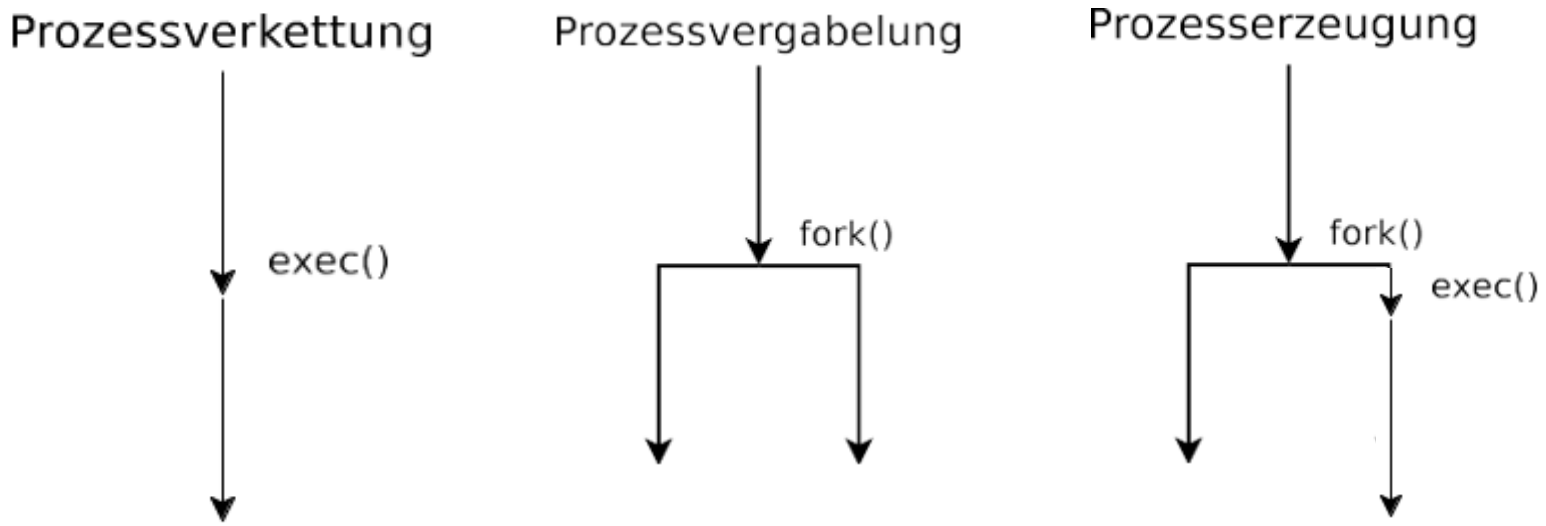
```
$ ./exec_beispiel
...Elternprozess...
[Eltern] Eigene PID:      25492
[Eltern] PID des Kindes: 25493
[25492 ]Programmende
...Kindprozess...
[Kind]   Eigene PID:      25493
[Kind]   PID des Vaters: 25492
Di 24. Mai 17:16:48 CEST 2016

$ ./exec_beispiel
...Elternprozess...
[Eltern] Eigene PID:      25499
[Eltern] PID des Kindes: 25500
[25499 ]Programmende
...Kindprozess...
[Kind]   Eigene PID:      25500
[Kind]   PID des Vaters: 1
Di 24. Mai 17:17:15 CEST 2016
```

- Der Kindprozess wird nach der Ausgabe seiner PID mit `getpid()` und der PID seines Elternprozesses mit `getppid()` durch `date` ersetzt
- Wird der Elternprozess vor dem Kindprozess beendet, bekommt der Kindprozess `init` als neuen Elternprozess zugeordnet

Seit Linux Kernel 3.4 (2012) und Dragonfly BSD 4.2 (2015) können auch andere Prozesse als PID=1 sich als potentielle neue Elternprozesse für verwaiste Kindprozesse anmelden
http://unix.stackexchange.com/questions/149319/new-parent-process-when-the-parent-process-dies/177361#177361

3 Möglichkeiten, einen (neuen) Prozeß zu erzeugen

- **Prozessverkettung** (*chaining*):
  Ein laufender Prozess ändert mit exec() seinen Prozesskontext und wird damit durch den neuen Prozess (mit gleicher PID und PPID) ersetzt
- **Prozessvergabelung** (*forking*):
  Ein laufender Prozess erzeugt mit fork() einen neuen, identischen Prozess
- **Prozesserzeugung** (*creation*): Ein laufender Prozess erzeugt mit fork() einen neuen, identischen Prozess, der sich selbst mit exec() durch einen neuen Prozess ersetzt



Spaß haben mit Forkbomben

- Eine Forkbombe ist ein Programm, das den Systemaufruf *fork()* in einer Endlosschleife aufruft
- Ziel: So lange Kopien des Prozesses erzeugen, bis kein Speicher mehr frei ist
  - Das System wird unbenutzbar

Forkbombe in Python

```
1 import os
2
3 while True:
4       os.fork()
```

Forkbombe in C

```
1 #include <unistd.h>
2
3 int main(void)
4 {
5       while(1)
6             fork();
7 }
```

Forkbombe in PHP

```
1 <?php
2 while(true)
3     pcntl_fork();
4 ?>
```

Einzige Schutzmöglichkeit:
Maximale Anzahl der Prozesse und maximalen Speicherverbrauch pro Benutzer limitieren

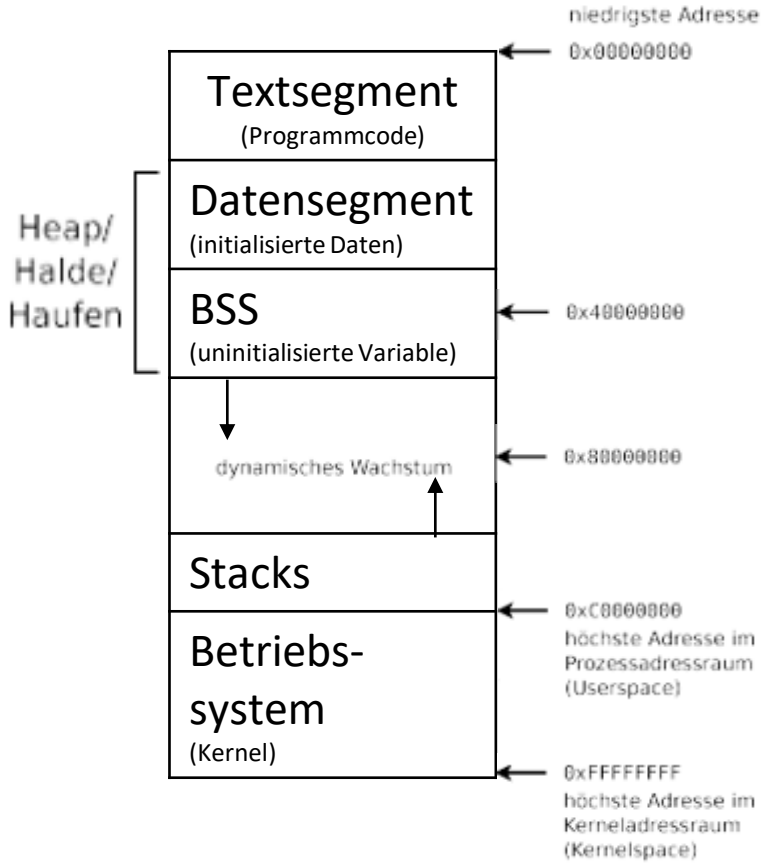
Struktur eines UNIX-Prozesses im Speicher (1/6)

Standardmäßige Aufteilung des virtuellen Speichers auf einem Linux-System mit 32-Bit-CPU:

- 1 GB sind für das System (Kernel)
- 3 GB für den laufenden Prozess

Die Struktur von Prozessen auf 64 Bit-Systemen unterscheidet sich nicht von 32 Bit-Systemen. Einzig der Adressraum ist größer und damit die mögliche Ausdehnung der Prozesse im Speicher

- Das **Textsegment** enthält den Programmcode (Maschinencode).
- Ihn können mehrere Prozesse teilen (muss also nur einmal im physischen Speicher vorgehalten werden)
  - Ist darum üblicherweise nur lesbar (*read only*)

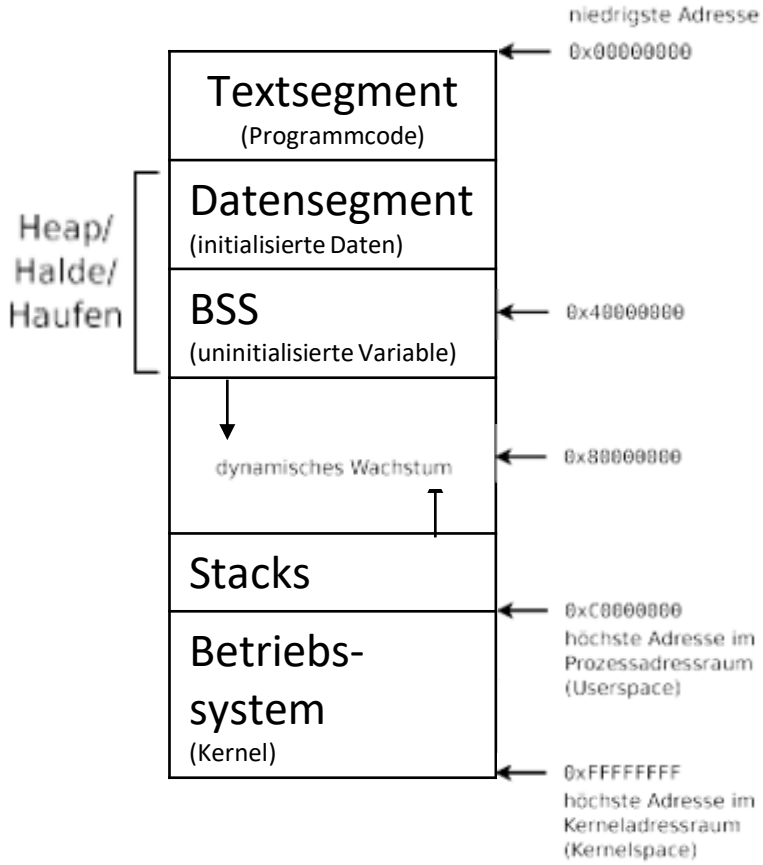


Struktur eines UNIX-Prozesses im Speicher (6/6)

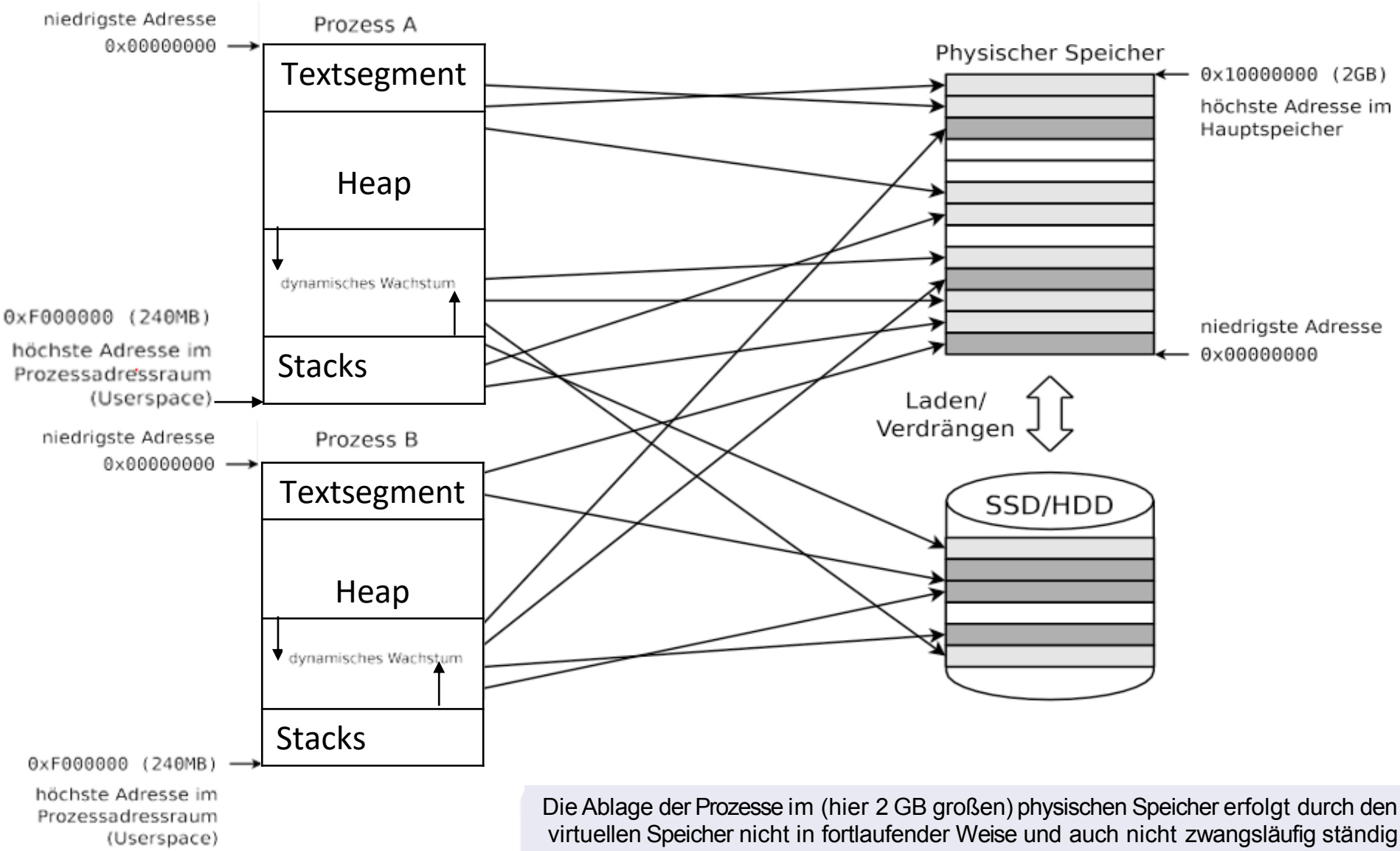
Das Kommando *size* gibt die Größe (in Bytes) von Textsegment, Datensegment und BSS von Programmdateien aus

- Die Inhalte von Textsegment und Datensegment sind in den Programmdateien enthalten
- Alle Inhalte im BSS werden bei der Prozesserzeugung auf den Wert 0 gesetzt

```
$ size /bin/c*                          Summe    Summe
   text    data     bss     dec     hex filename
  46480     620    1480   48580    bdc4 /bin/cat
   7619     420      32    8071    1f87 /bin/chacl
  55211     592     464   56267    dbcb /bin/chgrp
  51614     568     464   52646    cda6 /bin/chmod
  57349     600     464   58413    e42d /bin/chown
 120319     868    2696  123883   1e3eb /bin/cp
 131911    2672    1736  136319   2147f /bin/cpio
```



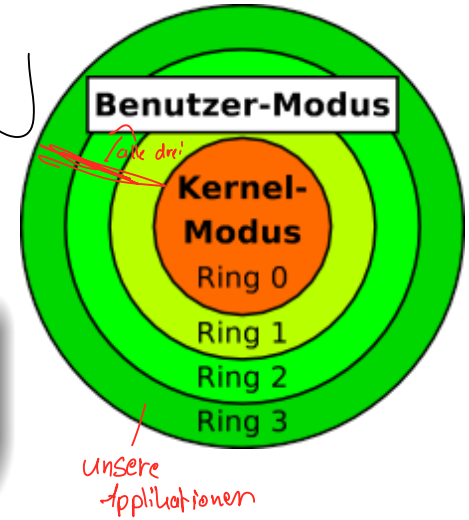
Virtueller Speicher



Die Ablage der Prozesse im (hier 2 GB großen) physischen Speicher erfolgt durch den virtuellen Speicher nicht in fortlaufender Weise und auch nicht zwangsläufig ständig im Hauptspeicher

Usermodus und Kernelmodus

- x86-kompatible CPUs enthalten 4 Privilegienstufen
  - Ziel: Stabilität und Sicherheit verbessern
  - Jeder Prozess wird in einem Ring ausgeführt und kann sich nicht selbstständig aus diesem befreien



Das Register CPL (Current Privilege Level) speichert die aktuelle Privilegienstufe

Quelle: Intel 80386 Programmer's Reference Manual 1986
http://css.csail.mit.edu/6.858/2012/readings/i386.pdf

In Ring 0 (= **Kernelmodus**) läuft der Betriebssystemkern

- Hier haben Prozesse vollen Zugriff auf die Hardware
- Der Kern kann auch physischen Speicher adressieren (⇒ Real Mode)

In Ring 3 (= **Benutzermodus**) laufen die Anwendungen

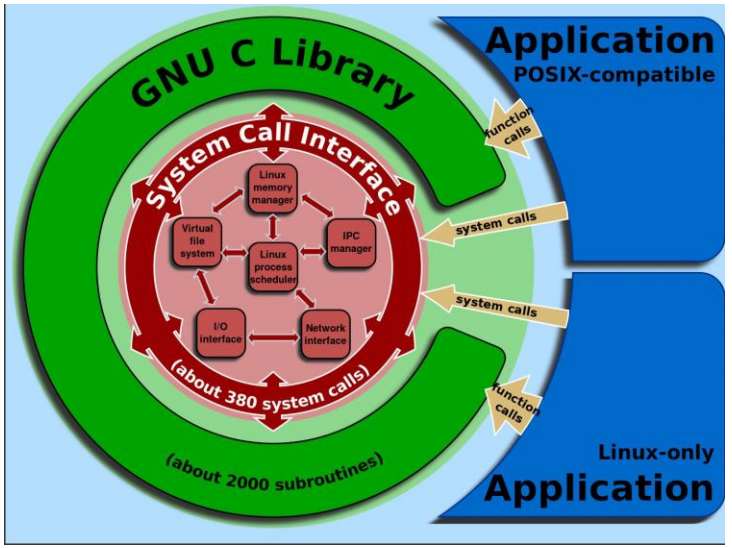
- Hier arbeiten Prozesse nur mit virtuellem Speicher (⇒ Protected Mode)

Moderne Betriebssysteme verwenden nur 2 Privilegienstufen (Ringe 0 und 3). Bsp. auf Ring 1: Gast-OS in der Virtual box
Grund: Einige Hardware-Architekturen (z.B: Alpha, PowerPC, MIPS) enthalten nur 2 Stufen

Systemaufrufe und Bibliotheken

- Direkt mit Systemaufrufen arbeiten ist **unsicher** und **schlecht portabel**
- Moderne Betriebssysteme enthalten eine Bibliothek, die sich logisch zwischen den Benutzerprozessen und dem Kern befindet

Beispiele für solche Bibliotheken:
GNU C-Bibliothek glibc (Linux),
C Standard Library (UNIX),
C Library Implementation (BSD),
Native API `ntdll.dll` (Windows)



Bildquelle: Wikipedia
(Shmuel Csaba Otto Traian, CC-BY-SA-3.0)

Die Bibliothek ist zuständig für:

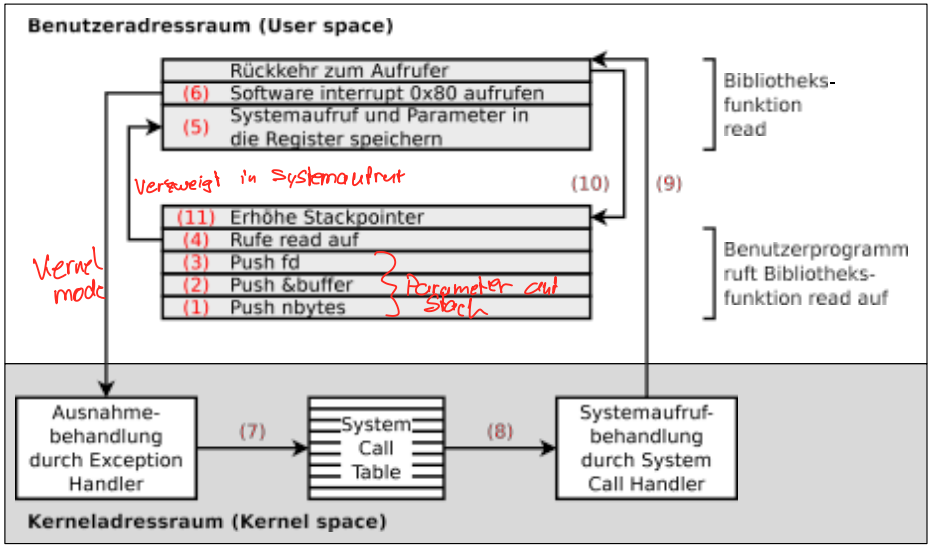
- Kommunikationsvermittlung der Benutzerprozesse mit dem Kernel
- Moduswechsel zwischen Benutzermodus und Kernelmodus

Vorteile, die der Einsatz einer Bibliothek mit sich bringt:

- Erhöhte **Portabilität**, da kein oder nur sehr wenig Bedarf besteht, dass die Benutzer-prozesse direkt mit dem Kernel kommunizieren
- Erhöhte **Sicherheit**, da die Benutzerprozesse nicht selbst den Wechsel in den Kernelmodus durchführen können

System call Schritt für Schritt (1/4) – read (fd, buffer, nbytes);

- In Schritt 1-3 legt der Benutzerprozess die Parameter auf den Stack
- In 4 ruft der Benutzerprozess die **Bibliotheksfunktion** für read (⇒ nbytes aus der Datei fd lesen und in buffer speichern) auf

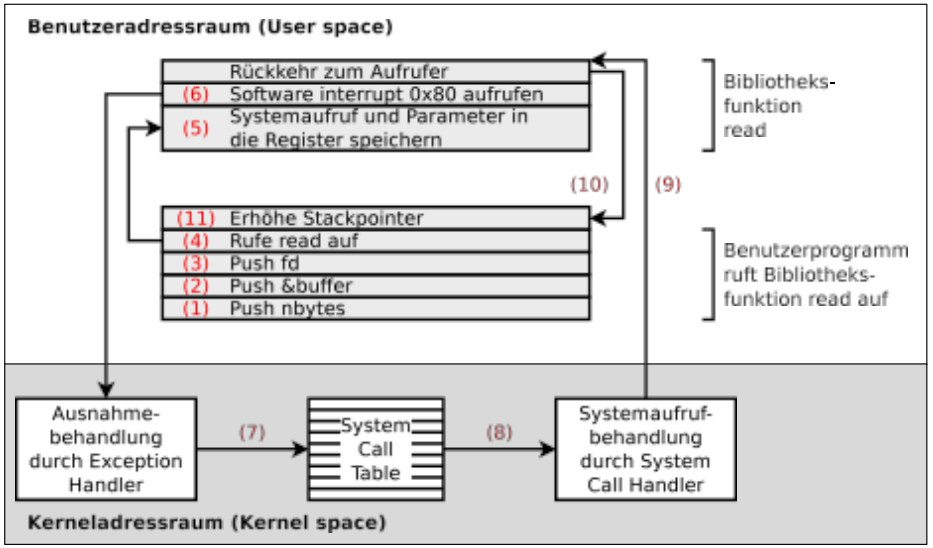


- Die Bibliotheksfunktion speichert in 5:
  - die Nummer des Systemaufrufs im Register EAX (*3*, 32-Bit) bzw. RAX (*0*, 64-Bit) (https://de.wikipedia.org/wiki/Liste_der_Linux-Systemaufrufe)
  - die Parameter des Systemaufrufs in den Registern EBX, ECX und EDX (bzw. bei 64-Bit: RBX, RCX und RDX)

Quelle dieses Beispiels
Moderne Betriebssysteme, *Andrew S. Tanenbaum*, 3.Auflage, Pearson (2009), S.84-89

System call Schritt für Schritt (2/4) – read (fd, buffer, nbytes);

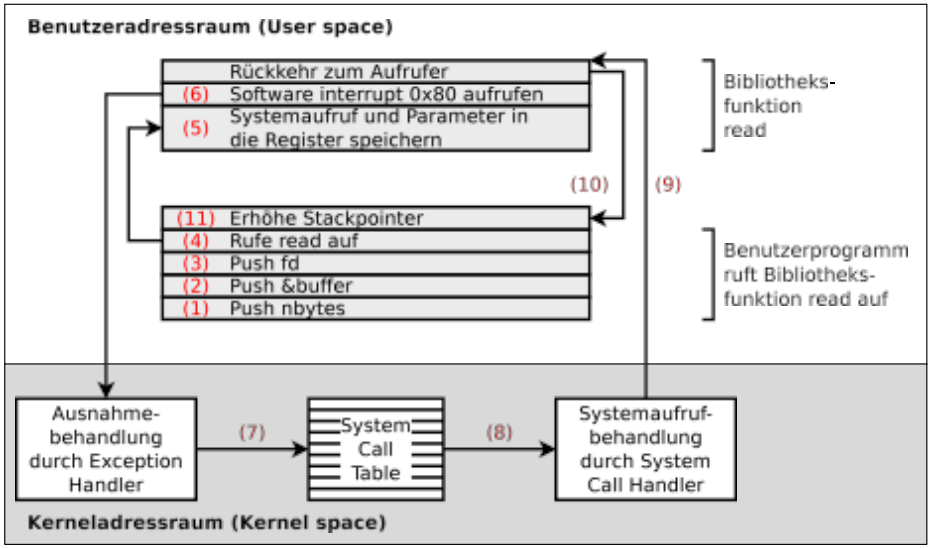
- In **6** wird der Softwareinterrupt (Exception) 0x80 (dezimal: 128) ausgelöst, um vom Benutzermodus in den Kernelmodus zu wechseln
  - Der Softwareinterrupt unterbricht die Programmausführung im Benutzermodus und erzwingt das Ausführen eines Exception-Handlers im Kernelmodus



Der Kernel verwaltet die *System Call Table*, eine Liste mit allen Systemaufrufen.
Jedem Systemaufruf ist dort eine eindeutige Nummer und eine Kernel-interne Funktion zugeordnet

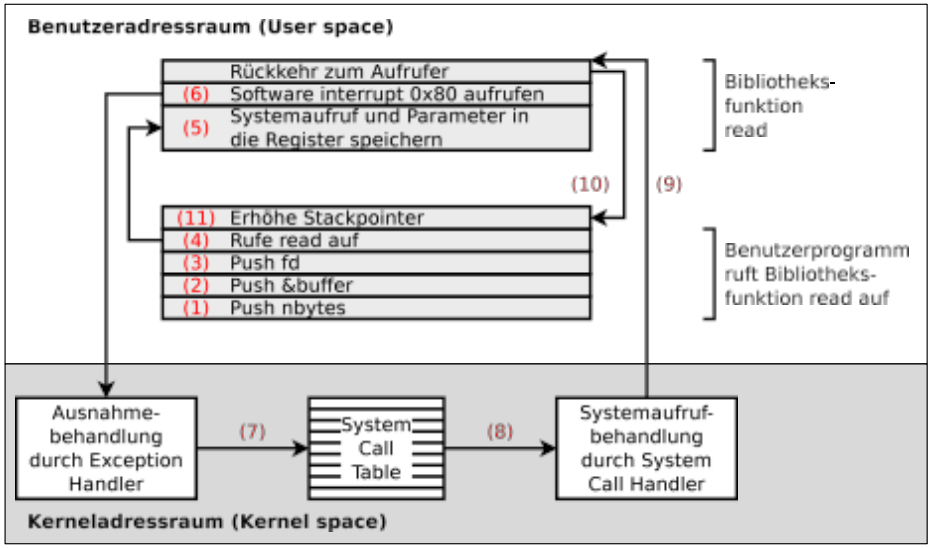
System call Schritt für Schritt (3/4) – read (fd, buffer, nbytes);

- Der aufgerufene Exception-Handler ist eine Funktion im Kernel, die das Register EAX ausliest
- Die Exception-Handler-Funktion ruft in **7** die entsprechende Kernel-Funktion aus der System Call Table mit den in den Registern EBX, ECX und EDX liegenden Argumenten auf
- In **8** startet der Systemaufruf



System call Schritt für Schritt (3/4) – read (fd, buffer, nbytes);

- In **9** gibt der Exception-Handler die Kontrolle an die Bibliothek zurück, die den Softwareinterrupt auslöste
- Die Funktion kehrt danach in **10** zum Benutzerprozess so zurück, wie es auch eine normale Funktion getan hätte



- Um den Systemaufruf abzuschließen, muss der Benutzerprozess in **11** genau wie nach jedem Funktionsaufruf den Stack aufräumen
- Anschließend kann der Benutzerprozess weiterarbeiten

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Linux System call Table (Ausschnitt)

x86_64 (64-bit)

Compiled from Linux 4.14.0 headers.

Nummer zu Systemcall

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count
2	open	man/ cs/	0x02	const char *filename	int flags	umode_t mode
3	close	man/ cs/	0x03	unsigned int fd	-	-
4	stat	man/ cs/	0x04	const char *filename	struct __old_kernel_stat *statbuf	-
5	fstat	man/ cs/	0x05	unsigned int fd	struct __old_kernel_stat *statbuf	-
6	lstat	man/ cs/	0x06	const char *filename	struct __old_kernel_stat *statbuf	-
7	poll	man/ cs/	0x07	struct pollfd *ufds	unsigned int nfds	int timeout
8	lseek	man/ cs/	0x08	unsigned int fd	off_t offset	unsigned int whence
9	mmap	man/ cs/	0x09	?	?	?
10	mprotect	man/ cs/	0x0a	unsigned long start	size_t len	unsigned long prot
11	munmap	man/ cs/	0x0b	unsigned long addr	size_t len	-
12	brk	man/ cs/	0x0c	unsigned long brk	-	-

Auswahl an Systemaufrufen

Prozess-verwaltung	`fork`	Neuen Kindprozess erzeugen
	`waitpid`	Auf Beendigung eines Kindprozesses warten
	`execve`	Einen Prozess durch einen anderen ersetzen. PID beibehalten
	`exit`	Prozess beenden
Datei-verwaltung	`open`	Datei zum Lesen/Schreiben öffnen
	`close`	Offene Datei schließen
	`read`	Daten aus einer Datei in den Puffer einlesen
	`write`	Daten aus dem Puffer in eine Datei schreiben
	`lseek`	Dateipositionszeiger positionieren
	`stat`	Status einer Datei ermitteln
Verzeichnis-verwaltung	`mkdir`	Neues Verzeichnis erzeugen
	`rmdir`	Leeres Verzeichnis entfernen
	`link`	Neuen Verzeichniseinträg (Link) auf eine Datei erzeugen
	`unlink`	Verzeichniseintrag löschen
	`mount`	Dateisystem in die hierarchische Verzeichnisstruktur einhängen
	`umount`	Eingehängtes Dateisystem aushängen
Verschiedenes	`chdir`	Aktuelles Verzeichnis wechseln
	`chmod`	Dateirechte für eine Datei ändern
	`kill`	Signal an einen Prozess schicken
	`time`	Sekunden seit dem 1. Januar 1970 („Unixzeit“) ausgeben

Übung

1. Was ist ein PCB? Welche mind. 3 wichtigen Informationen verwaltet er?

- Zustand
- Befehlszähler
- Stackpointer
- Systemkontext

Übung

2.1. Was unterscheidet einen User-Level-Thread vom Kernel-(scheduled-)Thread?

Fiber

https://stackoverflow.com/questions/15983872/difference-between-user-level-and-kernel-supported-threads

2.2. Welche Beziehungen treten auf?

Kernel Thread muss laufen, damit Fiber aktiv sein kann

Erzeuger

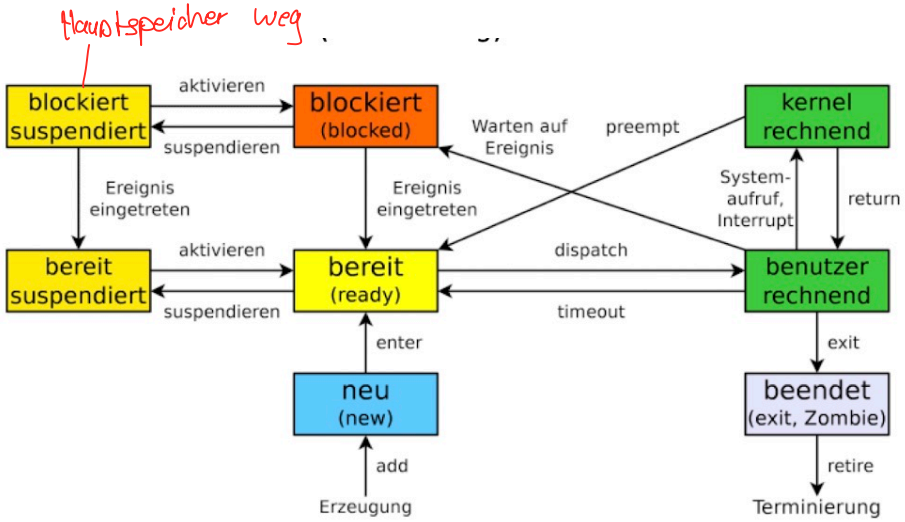
Übung

3. Was passiert im UNIX-Kommando fork()?

Prozess wird erzeugt

Übung

4. Welche Zustände kann ein Prozess einnehmen?



Übung

5. Wie entsteht ein Zombie-Prozess unter UNIX?

Parent beendet, hat aber auf Child nicht gewartet