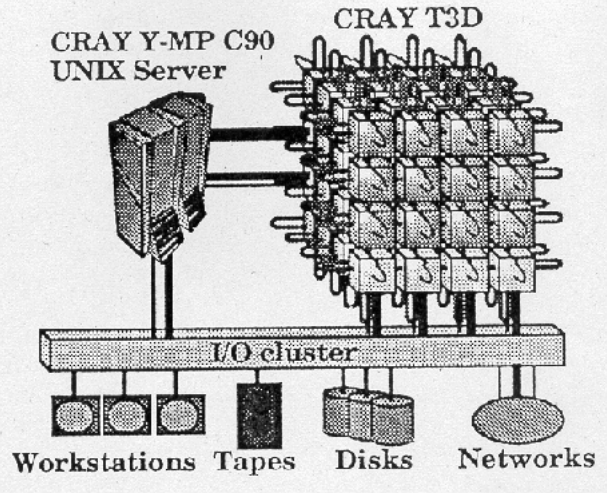
Betriebssysteme
Einleitung



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Literatur

Jürgen Nehmer, Peter Sturm:

Systemsoftware

- Verlag: Dpunkt Verlag; Auflage: 2., überarb. und aktualis. A. (März 2001)
- Sprache: Deutsch
- ISBN-10: 3898641155
- ISBN-13: 978-3898641159

Was macht ein Betriebssystem?

- Zweck
  - Selbstorganisation von Rechnersystemen
- Ansatz
  - Abstraktion von einer konkreten Hardware schaffen
    - Zahl der Prozessoren
    - Größe des Arbeitsspeichers
    - angeschlossene Geräte, z.B. Festplatte, Maus, Tastatur
- Ziel
  - Rechenleistung und Peripherie möglichst optimal ausnutzen
  - Programme weitgehend unabhängig von der unterliegenden Hardware einsetzen

Bestandteile eines Betriebssystems

- abstrakte Prozessor-Architektur
- Prozessverwaltung
  - Parallelität
  - abstrakte Prozessoren (engl. threads)
    - Nukleus
  - Prozesswechsel (engl. scheduling)
  - Prozessinteraktion
- Prozess-Synchronisation
  - nichtblockierende Verfahren
  - Semaphore
  - Monitore
  - Synchronisationsfehler

Bestandteile eines Betriebssystems

- Speicherverwaltung
  - starre Segmentierung
  - dynamische Segmentierung
- Adressräume
  - Seitenwechsel
  - virtuelle Adressierung
- Prozesskommunikation
  - Nachrichtenkommunikation
  - virtueller Speicher

Bestandteile eines Betriebssystems

- Ausnahmebehandlung
  - Signale
- Ein-/Ausgabe
  - abstrakte Geräte
- Betriebssystemkerne
  - Kernschnittstelle
  - Nukleusbasierte Kernimplementierung
- Dateiverwaltung

Entwicklung von Prozessorarchitekturen

- Unterscheidung von Prozessorarchitekturen
  - Einprozessorsysteme
  - Mehrprozessorsysteme
  - Mehrkernsysteme
    - homogen
    - heterogen

- Betriebssysteme liefern für Programme eine einheitliche Ablaufumgebung, weitgehend unabhängig von der Hardware.

Definition: Betriebssystem

- Unter dem Begriff Betriebssystem fasst man die Menge der Programme zusammen, die Benutzern die effiziente, gemeinsame Nutzung einer Rechenanlage (= Rechner) ermöglichen)
- auch definiert unter DIN 44300
- wichtigste Aufgaben
  - Annahme von Rechenaufträgen
  - Durchführung von Rechenaufträgen
  - langfristige Haltung von Daten und Programmen auf dafür geeigneten Datenträgern

Randbedingungen eines Betriebssystems

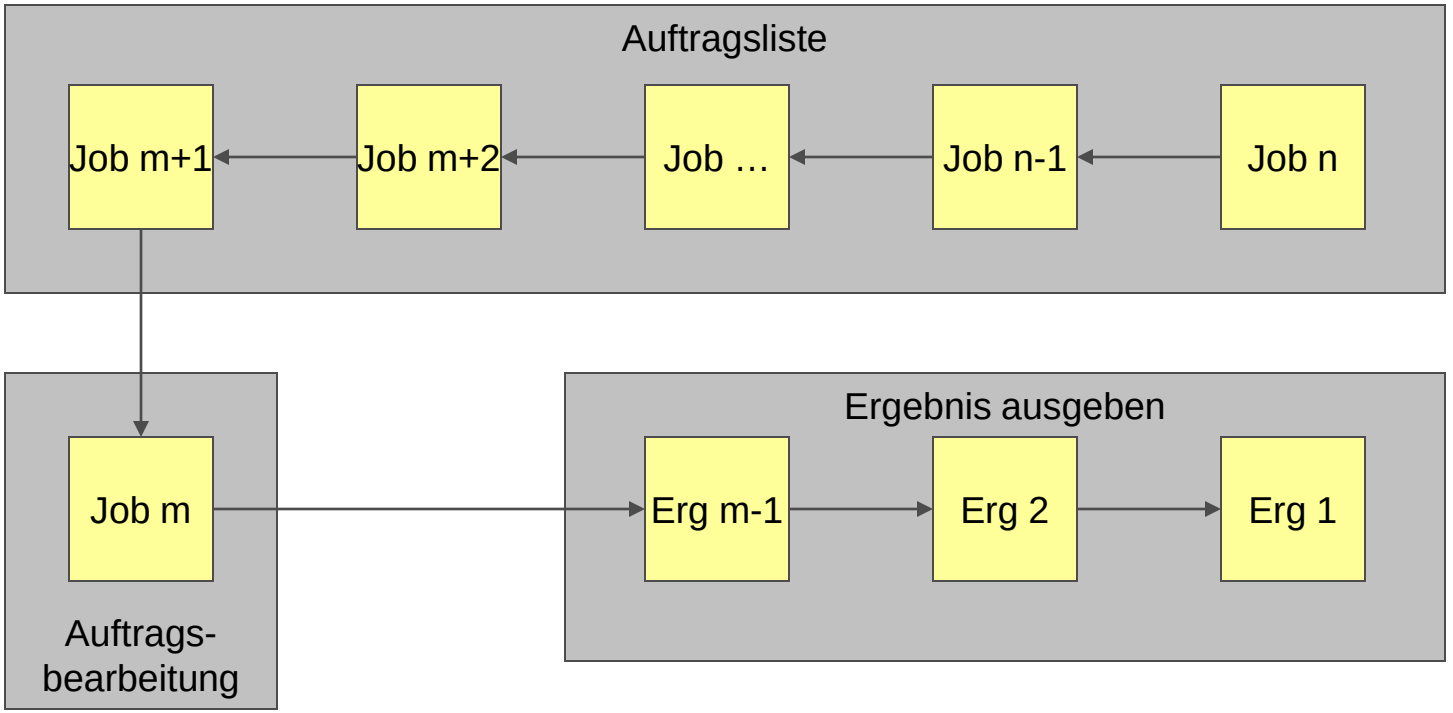
- Die vorhandenen Betriebsmittel des Rechners so einsetzen, dass Engpässe und Überlastsituationen vermieden werden.
- Das Betriebssystem sollte für die Durchführung seiner Aufgaben einen möglichst geringen Rechenaufwand benötigen.
  - engl. overhead
- Ein Betriebssystem muss robust gegen fehlerhafte Benutzerprogramme sein.
  - Programme durch geeignete Barrieren an der Zerstörung von Daten, Programmen und anderen Benutzerprogrammen hindern.
- Langfristig gehaltene Daten sollen gegen unerlaubten Zugriff und Ausfall von Datenträgern gesichert werden.

Arten von Betriebssystemen

- Ein reales Betriebssystem besteht aus Kompromissen.

- Betriebssysteme werden kategorisiert
  - Stapelbetrieb (engl. batch processing)
  - Dialogbetrieb (engl. time sharing)
  - Echtzeitbetrieb (engl. real time)

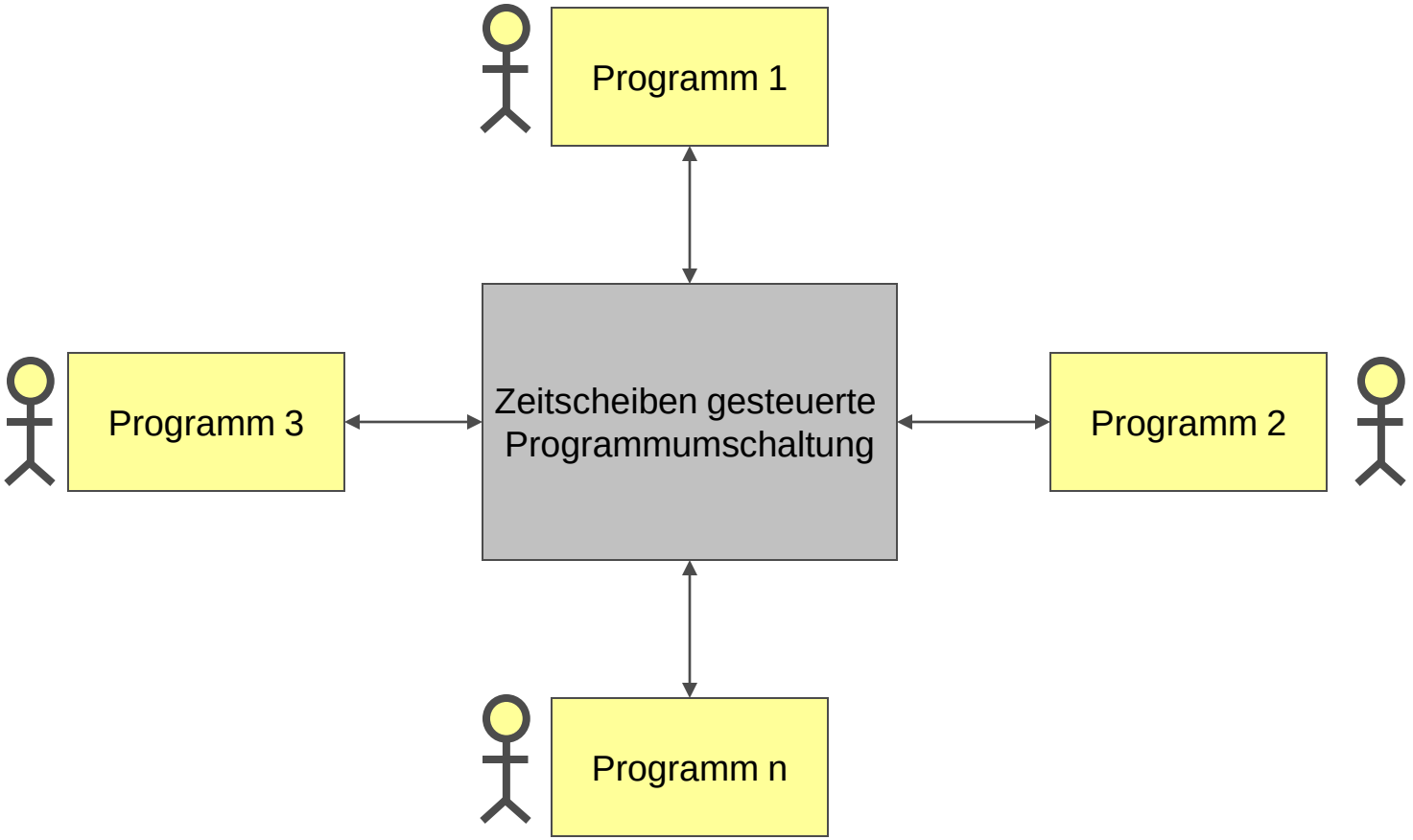
Prinzip: Stapelverarbeitung



Prinzip: Stapelverarbeitung

- Programme sind auf einem Datenträger sequentiell gespeichert.
- Betriebssystem führt die Programme (engl. jobs) nacheinander aus und übergibt die Ergebnisse an eine Ausgabe.
- Heute in allen Betriebssystemen für Berechnungen im Hintergrund vorhanden.
  - engl. batch jobs

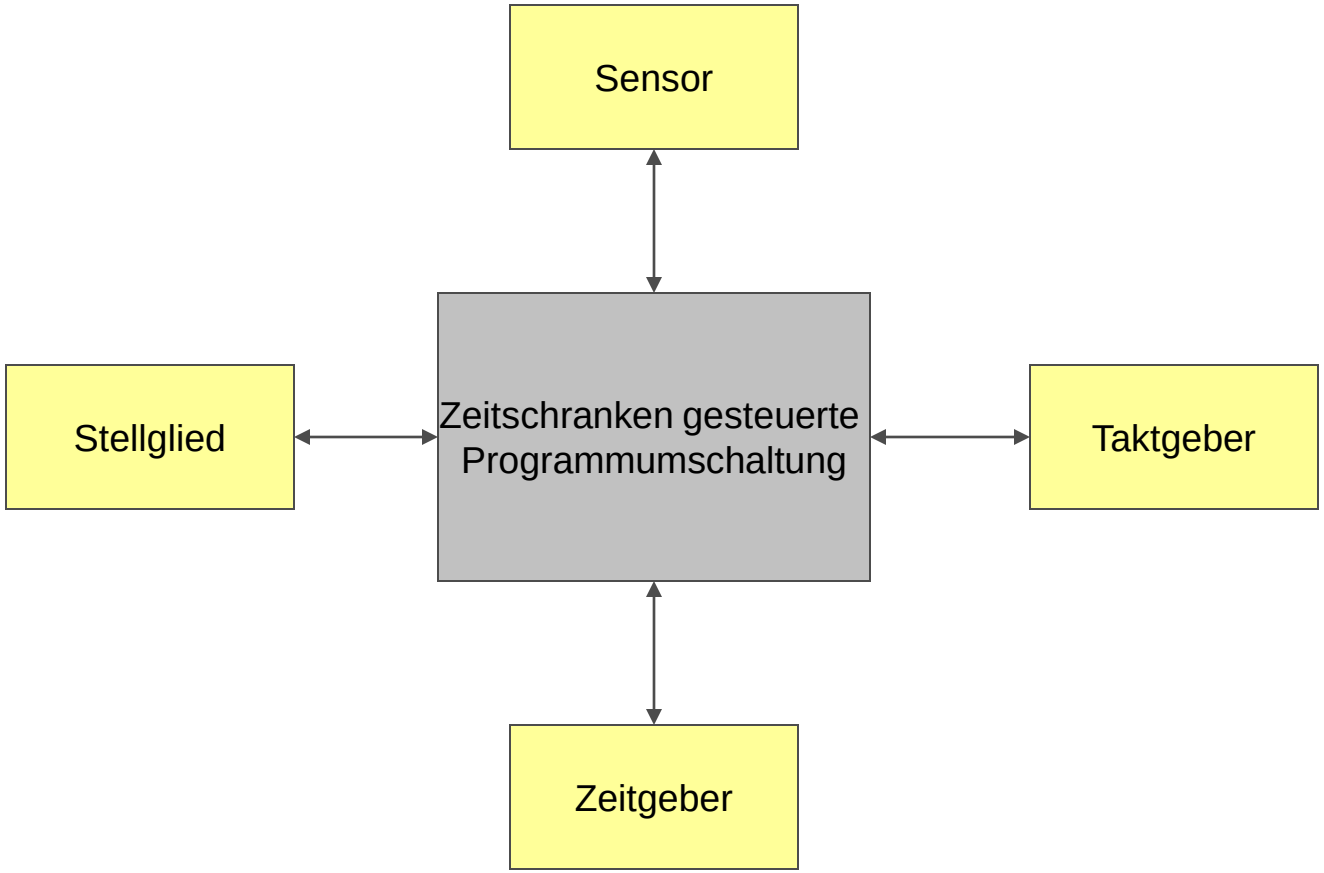
Prinzip: Dialogverarbeitung



Prinzip: Dialogverarbeitung

- Mehrere Benutzer nutzen den Rechner konkurrierend.
  - z.B. über ein Terminal
- Ein Terminal ist einem Benutzer für die Dauer der Sitzung fest zugeordnet.
- Alle Benutzer sollen den Eindruck haben, dass das Betriebssystem sofort auf ihre Eingabe reagiert.
  - wird bei Einprozessor-Systemen durch geeignete Verteilung von Zeitscheiben realisiert
  - heute Standard in dialogbasierten Betriebssystemen

Prinzip: Echtzeitverarbeitung

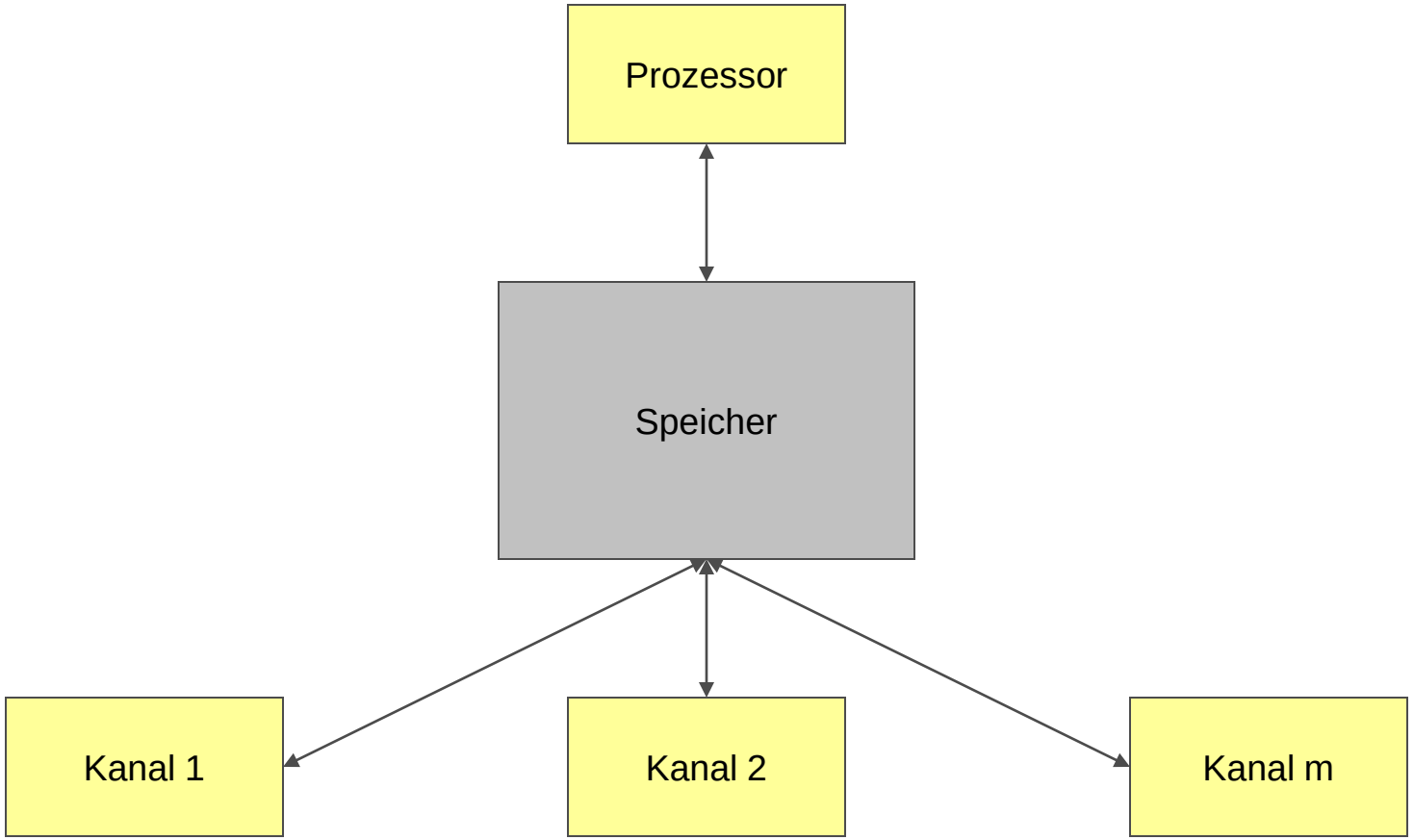


Prinzip: Echtzeitverarbeitung

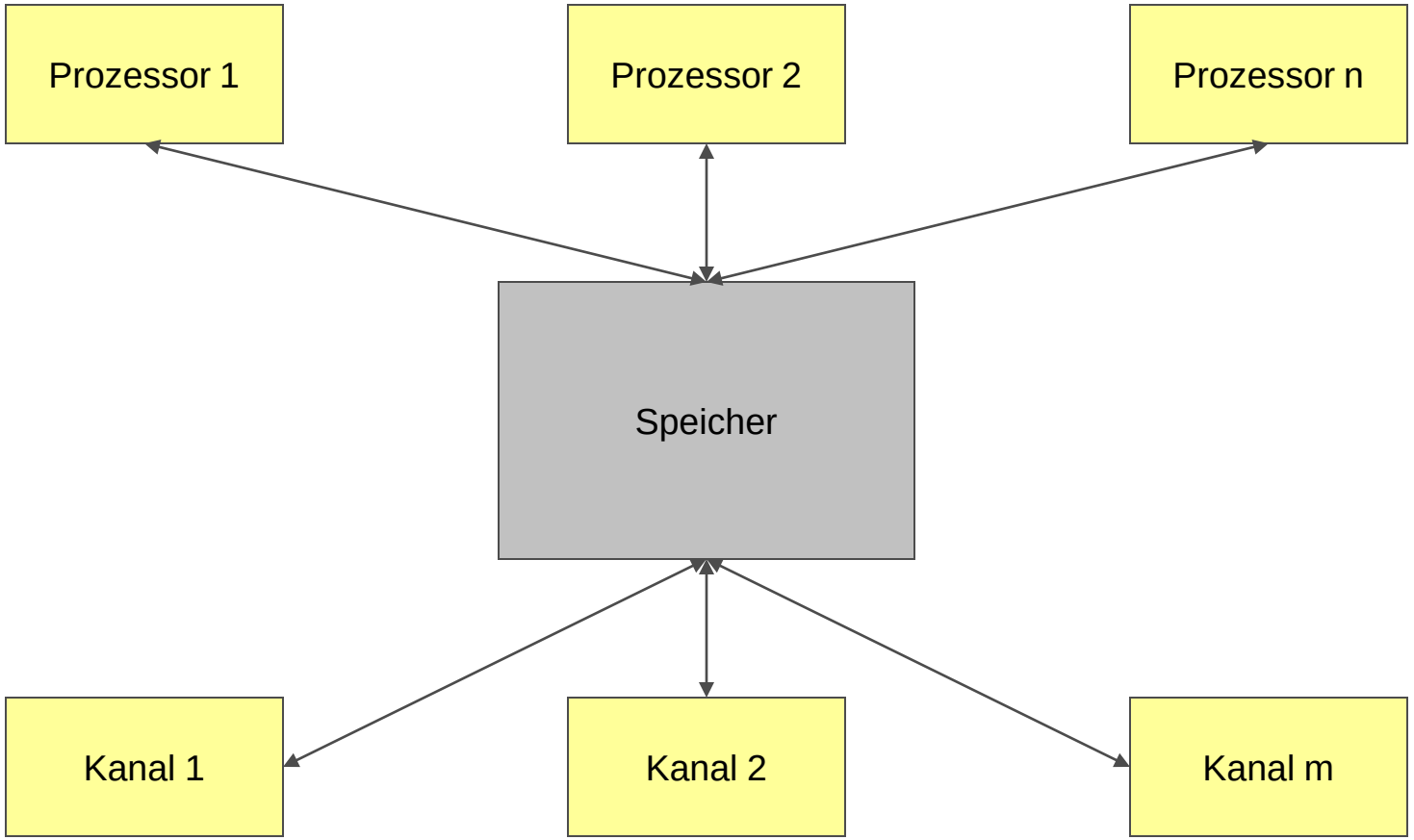
- Konkurrenz zwischen den Programmen durch jeweilig definierte Zeitbedingungen
  - engl. deadline

- Zeitbedingung legt fest, wann die Berechnung eines Programms (oder Programmteils) spätestens abgeschlossen sein muss.
  - harte Echtzeitbedingung
    - Nicht-Einhaltung führt zu materiellen Schäden oder kostet wahrscheinlich Menschenleben
  - weiche Echtzeitbedingung
    - Funktionsfähigkeit ist nicht gewährleistet
    - z.B. Fehlproduktion, Verlust von Daten oder Informationen

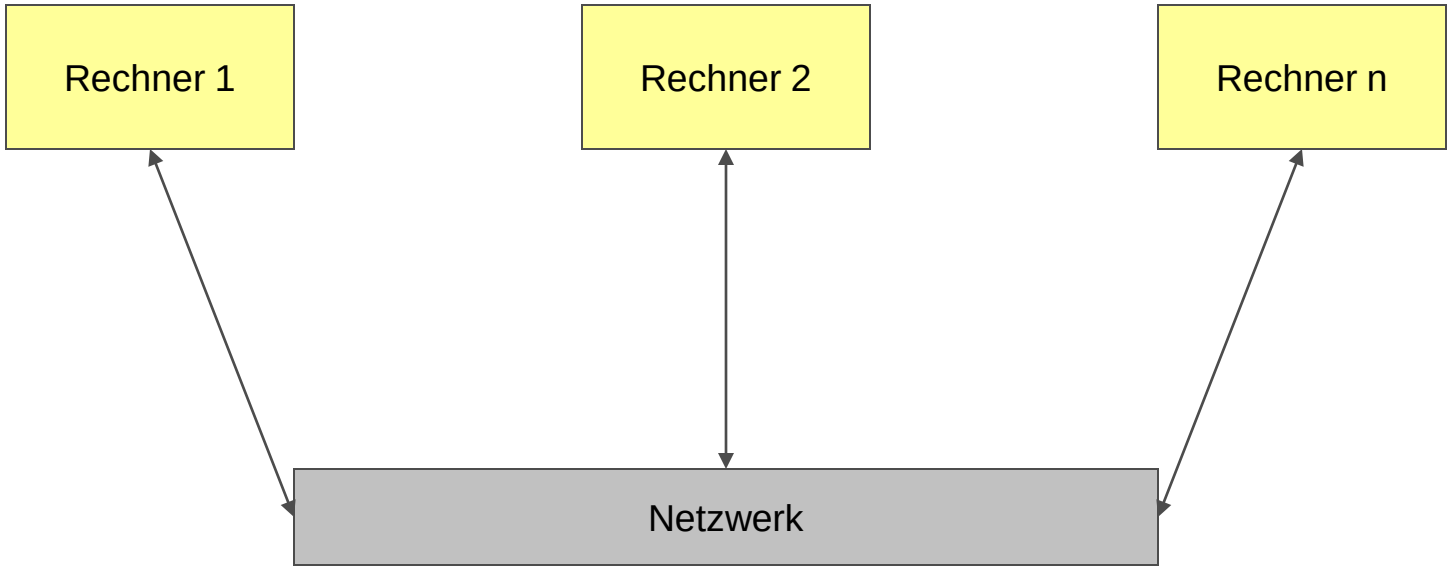
Einsatz: Monoprozessorkonfiguration



Einsatz: Mehrprozessorkonfiguration



Einsatz: Rechnernetz



Selbstkontrolle

1. Was optimiert ein Betriebssystem?
2. Welche Bestandteile besitzt ein Betriebssystem typischerweise?
3. Skizzieren Sie das Prinzip der Stapelverarbeitung.
4. Warum ist Stapelverarbeitung nicht für Dialogsysteme geeignet?
5. Sind Echtzeitbetriebssysteme nach Ihrer Meinung schneller als „herkömmliche“ Betriebssysteme?

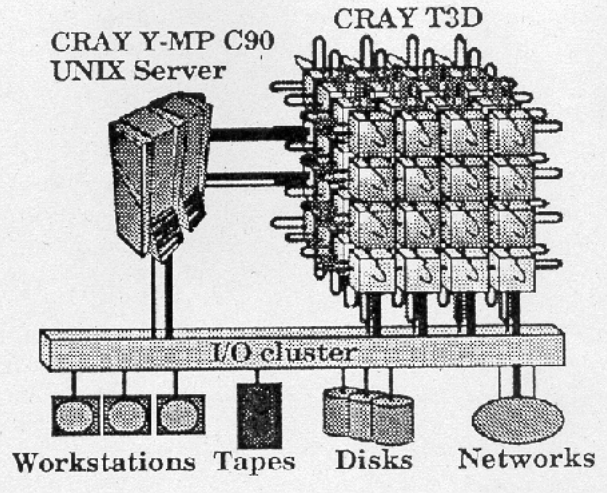
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Betriebssysteme
Architekturmodelle



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Kommandoschnittstelle vs. Call-Schnittstelle

- Architektur eines BS hängt vom Blickwinkel ab.
  - aus Sicht des Anwenders: Kommandoschnittstelle

- Kommandoschnittstelle
  - Kommandos zur Steuerung des Betriebssysteme
  - oft leicht austauschbar und für Anwenderbedürfnisse individuell anpassbar
  - nicht fest in das Betriebssystem integriert sondern einfaches Benutzerprogramm
  - oft sind mehrere Varianten einer Kommandoschnittstelle installiert
    - z.B. sh, bsh, csh, bash, ksh unter Linux

- CALL-Schnittstelle
  - definiert durch ein Benutzerprogramm aufrufbare Betriebssystem-Funktionen
  - komfortable Abstraktion, um Betriebsmittel des Rechners in Anspruch zu nehmen

CALL-Schnittstelle

- typischer Funktionssatz
  - Dateiverwaltung
  - Prozessorverwaltung
  - Prozesskoordinierung
  - Geräteverwaltung
  - Zeitdienst

- Abstraktion reduziert die Komplexität von Benutzerprogrammen.
  - weniger Aufwand
  - stabiler durch weniger Fehler

- Wesentliche Abstraktion heute: Beschreibung der Betriebssystemschnittstelle in einer Hochsprache
  - z.B. C++, C, etc.

CALL-Schnittstelle: Abstraktion durch Hochsprache

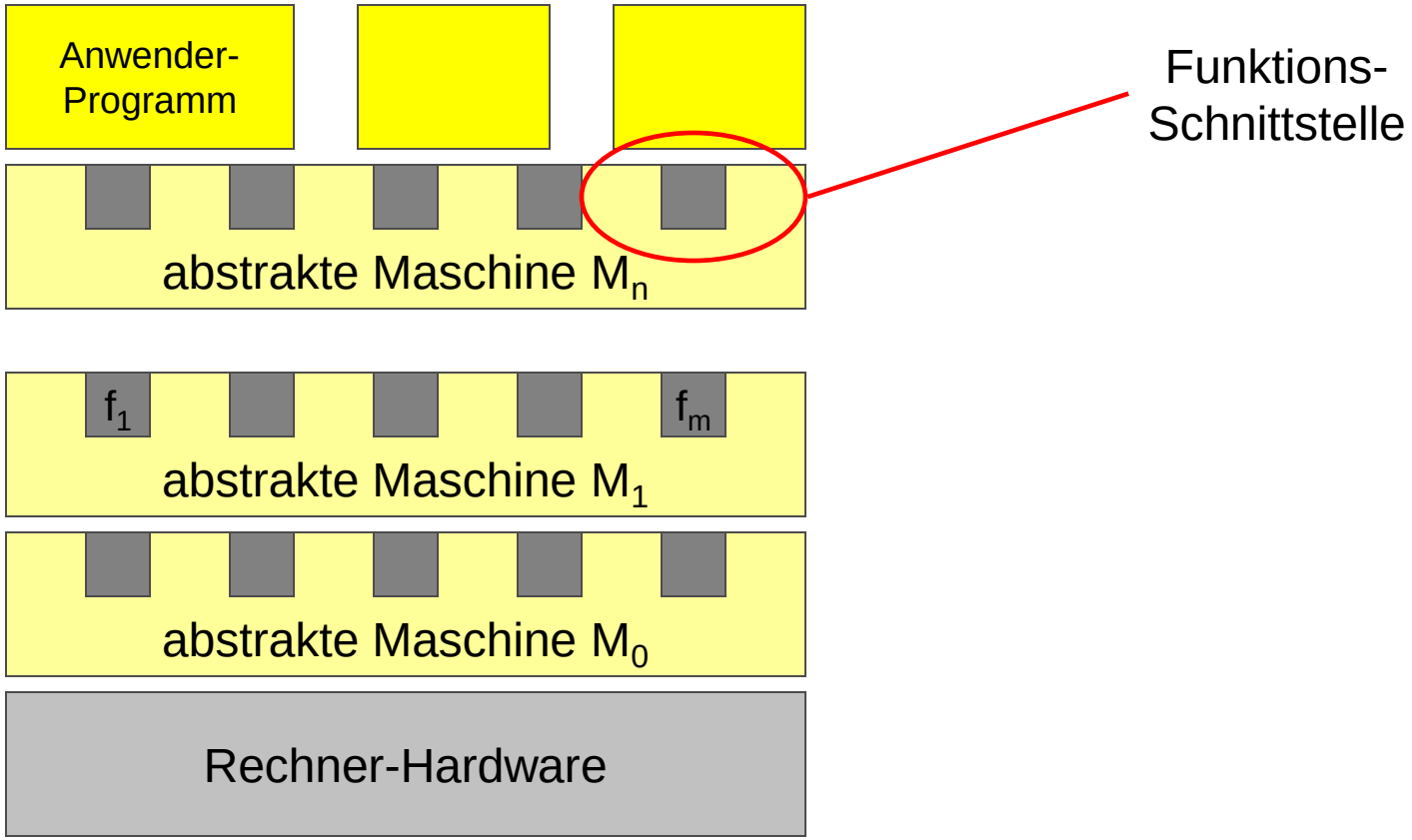
- Warum abstrahiert eine Hochsprache?
  - Einzelheiten der Rechnerarchitektur sind unwichtige Details bei der Beschreibung von Vorgängen des Betriebssystems.
    - z.B. Zahl der Register oder der Instruktionssatz des Prozessors
  - Hochsprache delegiert arithmetische und logische Operationen an einen Compiler oder Interpreter.
  - Die hochsprachliche Beschreibung von Algorithmen erleichtert die Portierung auf Rechner mit ähnlicher Architektur.

Strukturierung von Betriebssystemen: hierarchische Gliederung

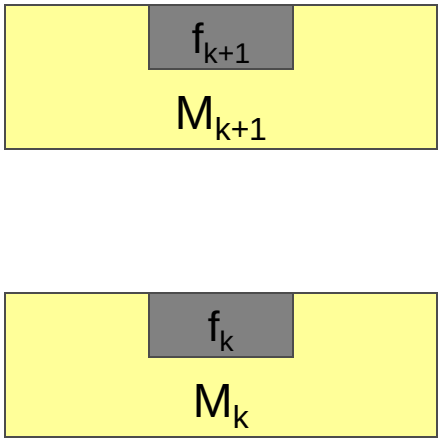
- Strukturierung in Schichten reduziert die Softwarekomplexität
- Schichten bauen aufeinander auf.
  - man spricht von einer „benutzt“-Relation
  - Benutzungsrichtung: von hardware-fern zu hardware-nah
  - Komplexität der Software wird kontrollierbar

- Jede Programmschicht bildet aus Sicht der nächst höheren Schicht eine abstrakte Maschine mit wohl definiertem Funktionssatz.
  - Maschineninstruktionen = Funktionssatz des physischen Rechners
  - Aufruf von Funktionen ist mit Methoden- oder Prozeduraufrufen vergleichbar.

Strukturierung von Betriebssystemen: hierarchische Gliederung

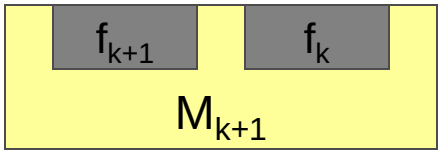


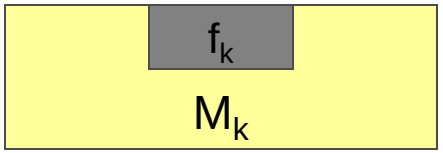
Varianten für hierarchische Gliederung: funktionelle Ersetzung



- Funktionssatz f_k der abstrakten Maschine M_k wird vollständig durch den Funktionssatz f_{k+1} der abstrakten Maschine M_{k+1} ersetzt.

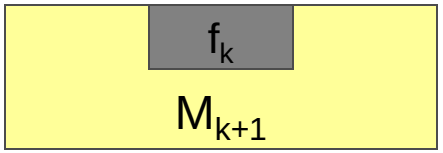
Varianten für hierarchische Gliederung: funktionelle Erweiterung

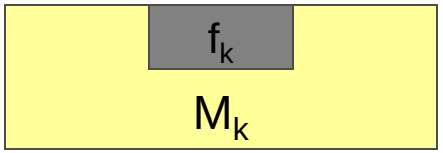




- Funktionssatz f_k der abstrakten Maschine M_k wird in der abstrakten Maschine M_{k+1} um den Funktionssatz f_{k+1} erweitert.
- Der Funktionssatz f_k bleibt dabei unverändert.

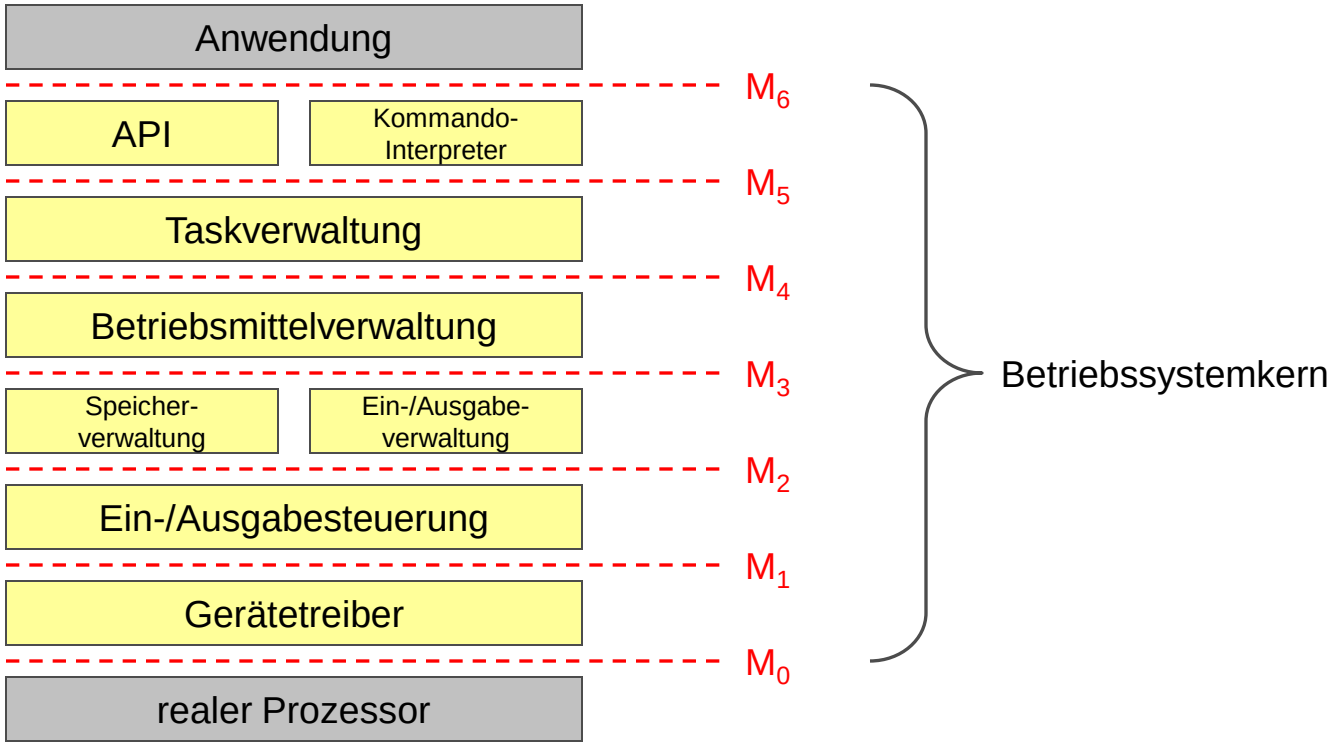
Varianten für hierarchische Gliederung: Virtualisierung





- Bei einem Virtualisierungsschritt bleibt die Funktionalität einer abstrakten Maschine M_k in M_{k+1} unverändert erhalten.
- In der Maschine M_{k+1} können nicht-funktionale Eigenschaften verändert werden.
  - Zuverlässigkeit
  - Robustheit
  - Performanz

Schichtenmodell eines Standard-Betriebssystems



Schichtenmodell eines Standard-Betriebssystems

- Gerätetreiber
  - Abstraktion von realer Hardware
- Ein-/Ausgabesteuerung
  - hardware-unabhängige Gerätesteuerung
- Ein-/Ausgabeverwaltung
  - Zuteilung von Geräten an Tasks
- Speicherverwaltung
  - Reservierung und Freigabe von Speicher

Schichtenmodell eines Standard-Betriebssystems

- Betriebsmittelverwaltung
  - generelle Zuteilung vom Betriebsmitteln
- Taskverwaltung
  - Zuteilung des Prozessors / der Prozessoren / der Kerne
- API (application programming interface)
  - Schnittstelle zu Anwenderprogrammen
- Kommandointerpreter
  - Eingabemöglichkeit für textuelle Befehle

Betriebssystemkern

- Unter einem Betriebssystemkern versteht man die Schichten, die kritische Aktionen in einem privilegierten Modus ausführen.
  - engl. kernel mode
  - nicht-privilegiert: Normalmodus (engl. user mode)

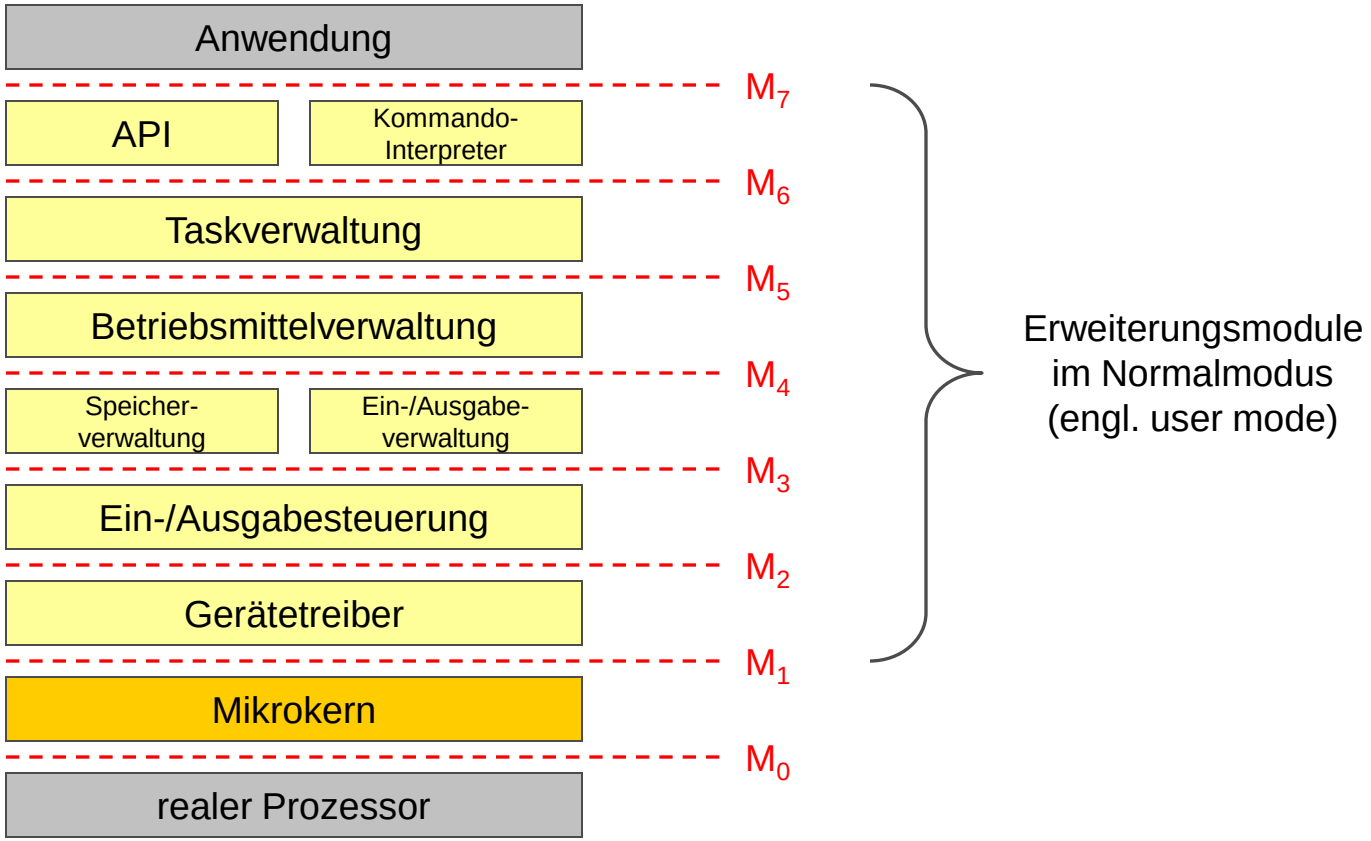
- Makrokern-Betriebssystem
  - Betriebssystem mit vielen Schichten im Kern
  - heute nicht mehr üblich

Mikrokern-Betriebssystem

- Ein Mikrokern-Betriebssystem besitzt einen kleinen Kern mit einer einzigen Schicht (engl. micro kernel)
  - alle unkritischen Aufgaben liegen außerhalb des Kerns
    - z.B. Scheduler
  - Weitere Schichten des Betriebssystems werden als Erweiterungsmodule im Normalmodus betrieben.

- Umfang eines Mirkokerns
  - Interprozess-Kommunikation
  - Synchronisation
  - elementare Funktionen der Taskverwaltung
    - einrichten
    - beenden
    - aktivieren
    - blockieren

Mikrokern-Betriebssystem



Mikrokern-Betriebssystem: Bewertung

- Vorteile
  - besser für eine Aufgabenstellung anpassbar als ein Makrokern-Betriebssystem
  - leichte Skalierbarkeit des Systems durch Hinzufügen oder Entfernen von Normalmodus-Modulen
  - Portierung auf andere Plattformen leichter
  - bessere zeitliche Vorhersagbarkeit
    - kritische Abläufe sind sehr kurz, nur eine Schicht beteiligt
    - kritische Abläufe dürfen nicht unterbrochen werden

Mikrokern-Betriebssystem: Bewertung

- Vorteile
  - im Optimalfall sind kritische Abläufe auf wenige Prozessor-Instruktionen beschränkt
    - die Annahme von Unterbrechungen muss nicht deaktiviert werden!

- Nachteile
  - geringerer Schutz des Systems durch Reduzierung der Erweiterungsmodule bei schwacher Rechenleistung
  - Performanzverlust durch häufiges Umschalten zwischen privilegiertem Modus und Normalmodus

Strukturierung bottom-up

- Bei der hierarchischen Strukturierung von Betriebssystemen geht man von der Hardwareseite vor (bottom-up).

- Pro abstrakter Maschine soll genau ein Konzept realisiert werden.

Beschreibung einer Abstraktion

- Eine Abstraktion in einer abstrakten Maschine M_k lässt sich auf folgende Weise darstellen:
  1. durch die Funktion f_k, die die abstrakte Maschine M_k bereitstellt
  2. durch eine Sprache l_k (engl. language), in der sich die abstrakten Konzepte widerspiegeln, die durch die abstrakte Maschine M_k unterstützt werden.

- Geht man davon aus, dass die Sprache l_{k+1} durch Funktionen f_{k+1} formuliert sind, lässt sich die Sprache l_{k+1} aus der Sprache l_k formulieren:

  $l_{k+1} \Leftrightarrow f_{k+1}(l_k)$

Warum hierarchische Strukturierung?

- Beherrschung der Vielfalt von Betriebssystemvarianten
  - jede abstrakte Maschine kann als Plattform für die Realisierung einer Familie von abstrakten Maschinen einer höheren Schicht dienen.
  - Entwicklung dedizierter Varianten von Betriebssystemen
    - auf lokales Umfeld spezialisiert
    - aus Standardbetriebssystem wird oft nur ein Teil der Leistungsfähigkeit benötigt

Client/Server Organisation in Betriebssystemen

- Idee: Funktionen in höhere Dienste gliedern
  - Verzeichnisdienst
  - Druckdienst
  - Mechanismen der Infrastruktur
- Mechanismen der Infrastruktur (Auswahl)
  - Prozesskommunikation
  - Prozessverwaltung

Client/Server Organisation in Betriebssystemen

- Alle Mechanismen der Infrastruktur werden zum Betriebssystemkern (kurz Kern, engl. kernel) zusammengefasst.
- Höhere Betriebssystemdienste werden als Menge (kooperierender) Server (Windows: Service) realisiert, die im Kern ablaufen.
- Client und Server laufen auf derselben Plattform (dem Kern) und unterliegen damit identischen Konstruktions-, Ablauf- und Interaktionsregeln.

Vorteile von Client-Server Organisation gegenüber reiner Schichtenarchitektur

- Betriebssystemdienste und Anwendung liegen auf dem gleichen Abstraktionsniveau
  - Entwicklung und Debuggen von Diensten kann aus der Anwendungsschicht erfolgen
- Client/Server Modell erzwingt mit Regeln und Schnittstellen eine durchgängige, einheitliche Modularisierung oberhalb des Kerns
  - Struktur gewährleistet einfache Erweiterbarkeit

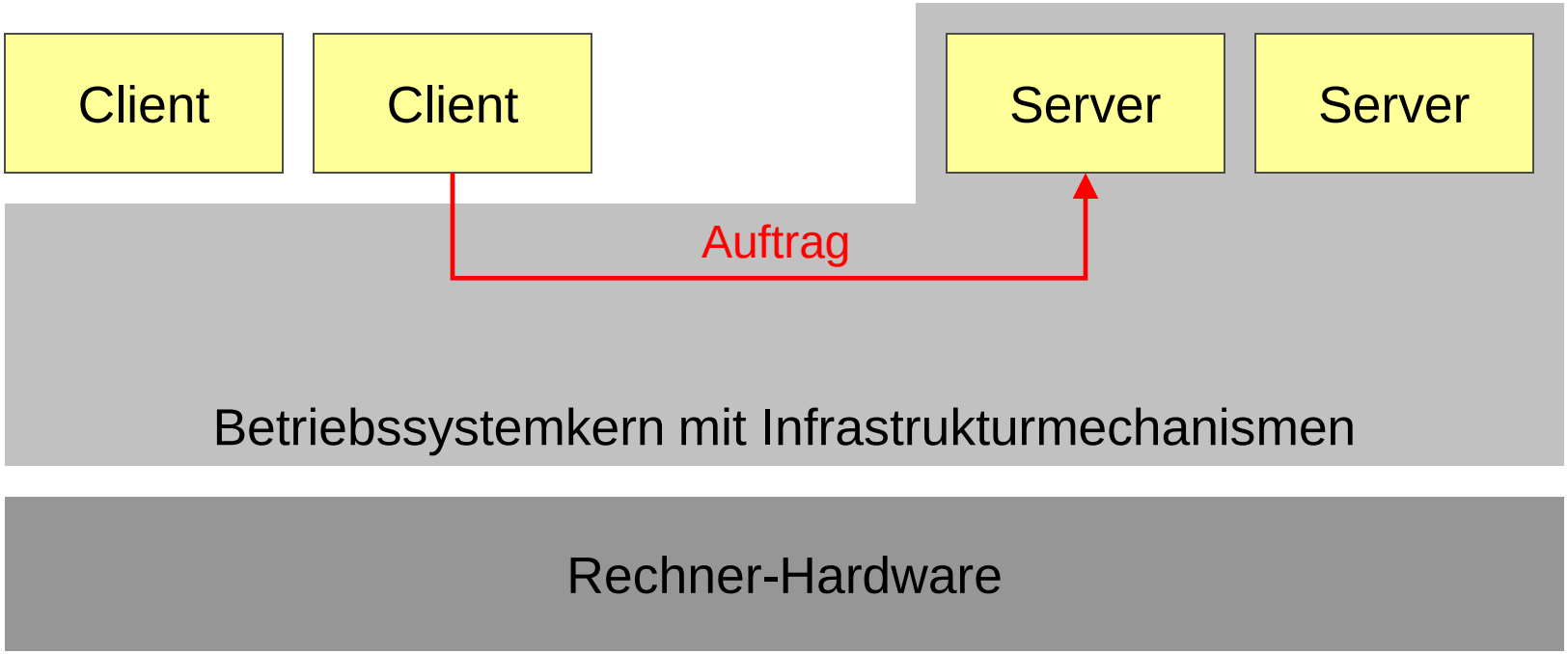
Vorteile von Client-Server Organisation gegenüber reiner Schichtenarchitektur

- Die Interaktion zwischen Anwendungen und Betriebssystem-Diensten, zwischen Anwendungskomponenten und Betriebssystem-Servern erfolgt einheitlich auf der Grundlage des Kommunikationsmechanismus, der durch den Kern bereitgestellt wird.

Vorteile von Client-Server Organisation gegenüber reiner Schichtenarchitektur

- Bei geeignetem Kommunikationsmechanismus können Client/Server Betriebssysteme auf vernetzten Rechnern ohne gemeinsamen Speicher ablaufen
  - Client/Server Organisation ist eine Verallgemeinerung der des Konzepts der virtuellen Maschine.
  - Virtualisierung von Maschinen erlaubt die Koexistenz mehrerer Betriebssysteme aus einem Rechner.
  - Moderne Prozessoren liefern bereits in der Hardwareschicht Funktionen zur Virtualisierung.
    - Ansatz: mehrere virtuelle Maschinen vom Typ des Wirtrechners (engl. host) erzeugen.

Client/Server Organisation

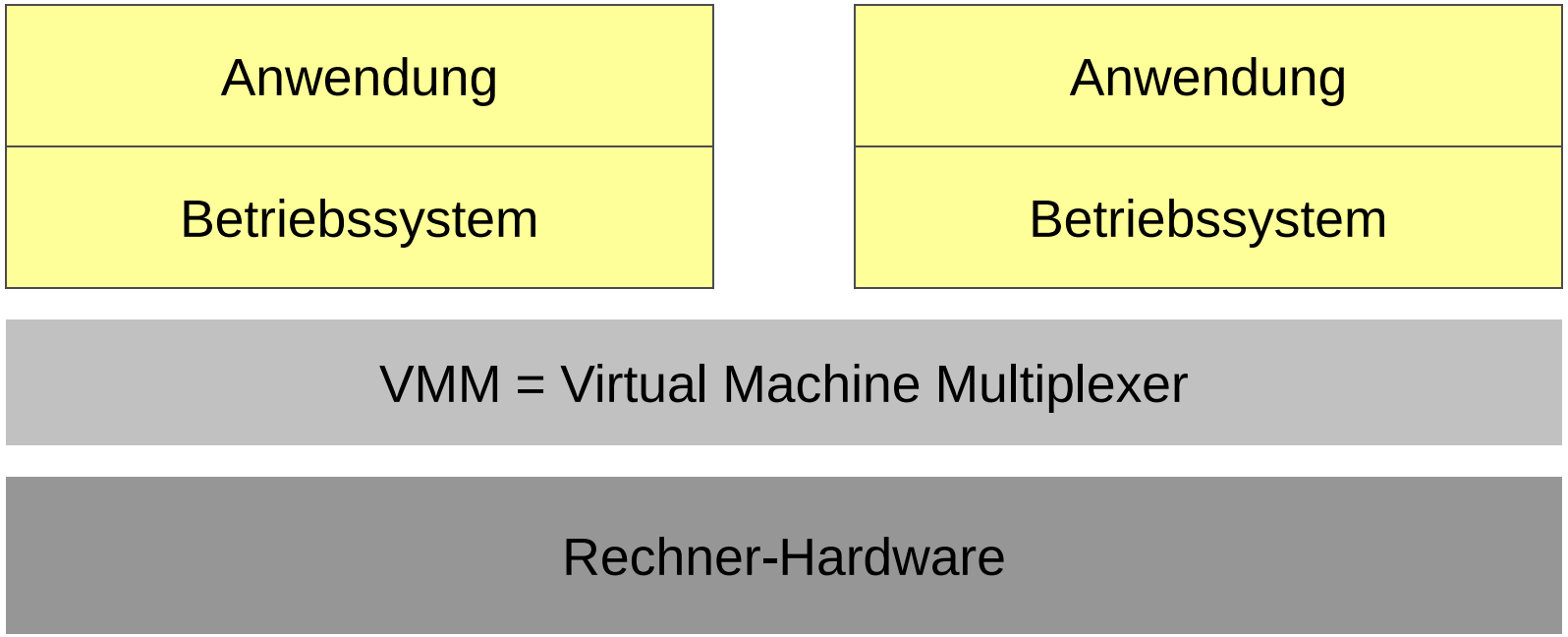


Virtualisierung in Client/Server Betriebssystemen

- Im Gegensatz zur Virtualisierung in einer Schichtenarchitektur können virtuelle Maschinen in einem Client/Server Betriebssystem beliebige Wirtrechner emulieren.
  - Prozessor
  - Speicher
  - Grafikkarte
  - Festplatte
- Virtualisierung wird heute als Anwenderprogramm installiert.

- typische Vertreter für Virtualisierungssoftware
  - Microsoft Virtual PC, Hyper-V
  - VM Ware
  - Virtual Box

Virtualisierung in Client/Server Betriebssystemen



Selbstkontrolle

1. Was unterscheidet eine Kommando-Schnittstelle von einer CALL-Schnittstelle?
2. Warum eignen sich Hochsprachen (C++, C#, Java) als Schnittstelle zum Betriebssystem?
3. Warum erfolgt die Benutzung von Schichten im Betriebssystem von hardware-fern zu hardware-nah?
4. Welche Schichten müssen Sie bei der Portierung eines Betriebssystems auf eine neue Hardware ändern?
5. Warum kann Virtualisierung zwischen zwei Schichten sinnvoll sein?
6. Was ist ein Kernel?
7. Warum eignet sich Client/Server Organisation für die Virtualisierung von Rechnern?

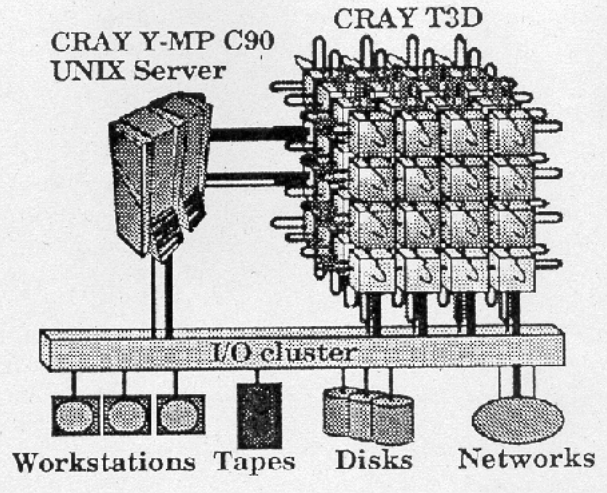
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Betriebssysteme
Abstrakte Prozessorarchitektur



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Warum eine abstrakte Prozessorarchitektur?

- Moderne Prozessoren unterscheiden sich erheblich
  - Aufbau der Instruktionen
  - Anzahl und Semantik von Instruktionen
  - Zahl und Zweck von Registern
  - Unterbrechungsbehandlung
  - Ein-/Ausgabe-Konzept

- Betriebssystemkonzepte werden auf einer Abstraktion realer Prozessoren definiert.
  - Definition des Funktionssatzes der abstrakten Maschine M_0.
  - M_0 liefert die Grundlage für spätere hardwarenahe Betriebssystem-Mechanismen.

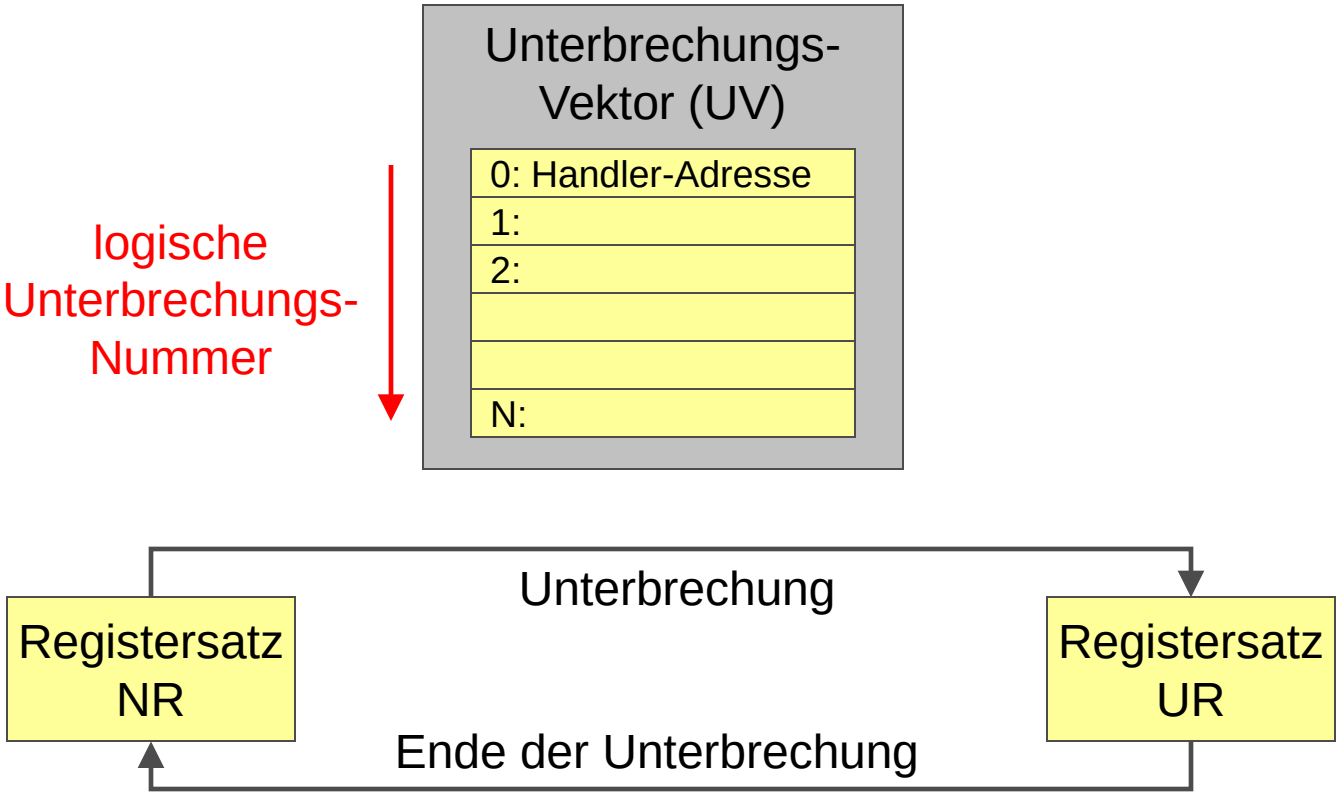
Ein abstrakter Prozessor: Registerstruktur und Unterbrechungsbehandlung

- Elemente eines abstrakten Prozessors
  - Registersatz im Normalmodus (NR)
  - Registersatz im Unterbrechungsmodus (UR)
  - Unterbrechungsvektor (UV)

- Privilegierte Instruktionen sollten aus Schutzgründen nur Betriebssystem-Programmen erlaubt sein.

- privilegierte Instruktionen (Auswahl)
  - Ein-/Ausgabe-Instruktionen
  - Änderung des Prozessorzustands

Registerstruktur eines abstrakten Prozessors



Unterbrechungen

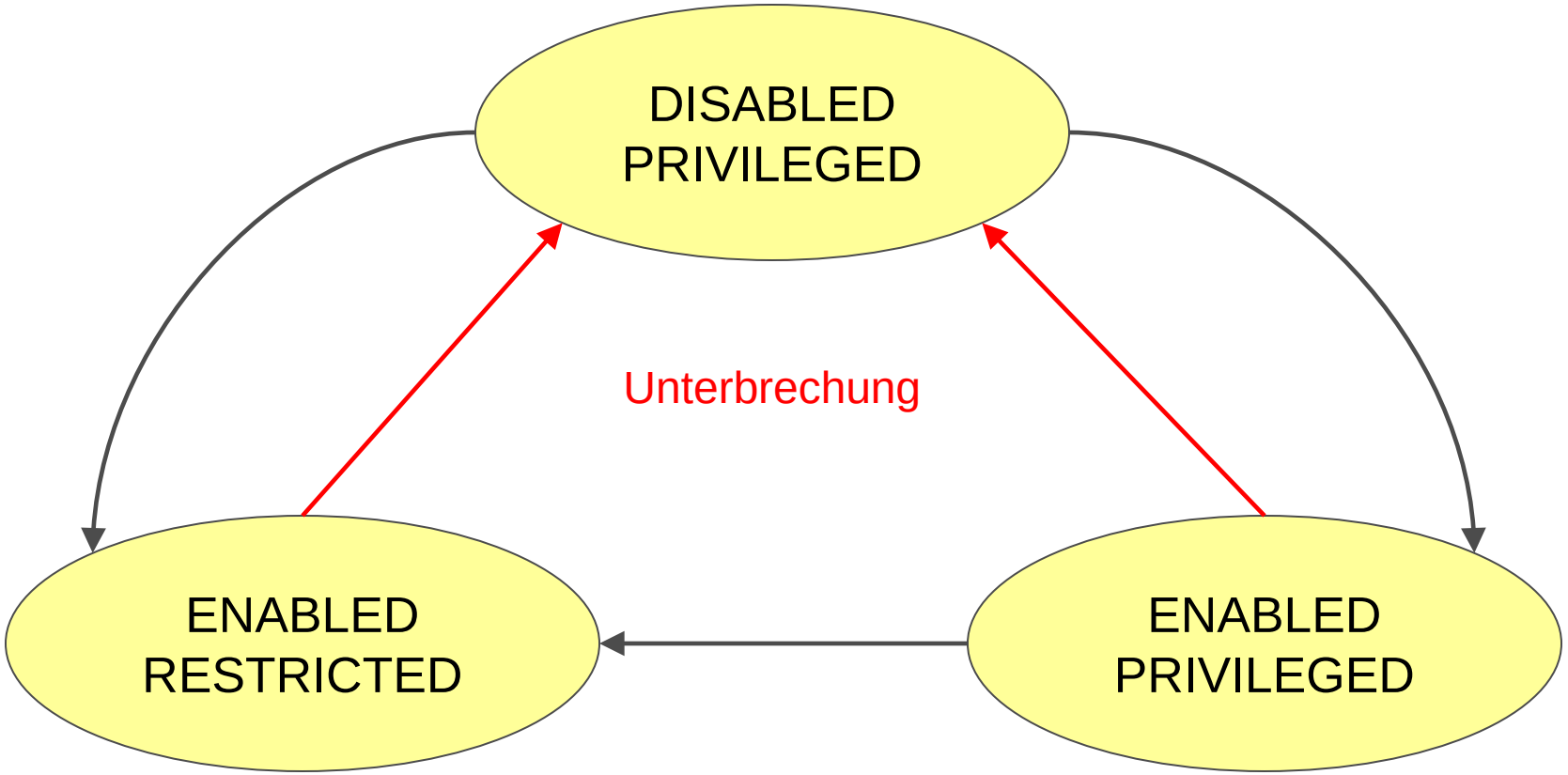
- Ursachen für Unterbrechungen werden in einem Unterbrechungsvektor UV definiert.
  - In jedem Eintrag des Vektors steht die Startadresse eines Programms zur Unterbrechungsbehandlung (engl. interrupt handler).
  - Der Index des Vektors wird als Unterbrechungsnummer bezeichnet.

- Ursachen für Unterbrechungen
  - Maschinenfehler
  - Ausnahmen (engl. exception), z.B. arithmetischer Überlauf
  - Trap-Befehl (Unterbrechung aus einem Programm heraus)

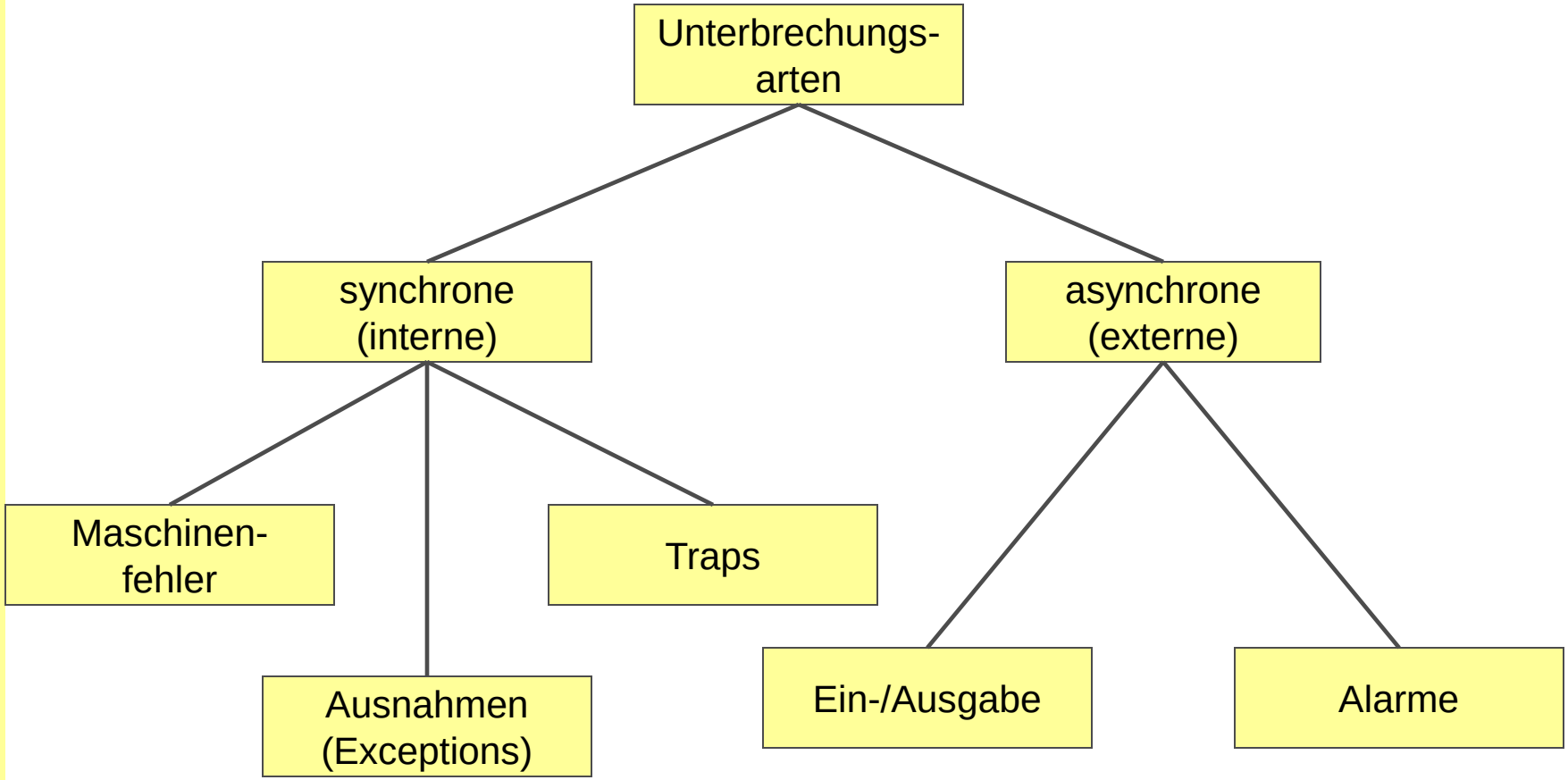
Prozessorzustände und Übergänge

- Prozessorzustände
  - ENABLED_RESTRICTED
    - Normalmodus, Prozessor arbeitet auf NR
    - Befehlssatz ist eingeschränkt (RESTRICTED): Ausführung privilegierter Operationen ist nicht erlaubt
  - ENABLED_PRIVILEGED
    - Prozessor nimmt Unterbrechungen an (ENABLED)
    - Zustand zur schnelleren Bearbeitung von Unterbrechungen
  - DISABLED_PRIVILEGED
    - Unterbrechungsmodus, Prozessor ist gesperrt gegen weitere Unterbrechungen
    - Eintreffende Unterbrechungen liegen bis zur Aufhebung der Sperre an

Prozessorzustände und Übergänge



Klassifizierung von Unterbrechungen



Annahme eines Unterbrechungssignals

- Annahme des Unterbrechungssignals i löst folgende unteilbare Aktion aus:

1. Prozessorzustand (UR) = DISABLED_PRIVILEGED
2. Parameter i auf Laufzeitkeller (UR) ablegen
3. aktiver Registersatz = UR
4. Unterprogrammsprung nach UV[i]

Rücksprung in den Normalzustand

- Rücksprung aus einem Unterbrechungs-Programm löst folgende unteilbare Aktion aus:

1. aktiver Registersatz = NR
2. Fortsetzung der Programmausführung

Behandlung von Unterbrechungen

- Die Unterbrechungsbehandlung funktioniert aus der Sicht eines Programmierers wir ein gewöhnliches Programm.
- Unterbrechungsbehandlung muss mindestens zwei Funktionen besitzen:
  - Behandlung eines Interrupts deaktivieren.
    - eintreffende Interrupts bleiben wirkungslos.
  - Behandlung eines Interrupts aktivieren.
    - eintreffende Interrupts werden wieder behandelt.

Methoden der Prozessverwaltung

- Unterbrechungsbehandlung muss den Zustand (Kontext) eines Prozesses speichern und wiederherstellen können.
  - Zustand speichern bei Eintreffen eines Interrupt
  - Zustand wiederherstellen bei Rücksprung aus der Unterbrechungsbehandlung

Ein abstrakter Prozessor: Ein-/Ausgabe

- Für Ein-/Ausgabe gibt es zwei gegensätzliche Konzepte
  - Inhalt eines Registers direkt an eine Ein-/Ausgabe-Schnittstelle übergeben bzw. lesen (prozessorintegrierte Ein-/Ausgabe)
  - Ein-/Ausgabe-Gerät als einen Speicherbereich betrachten (speicherintegrierte Ein-/Ausgabe)

Speicherintegrierte Ein-/Ausgabe

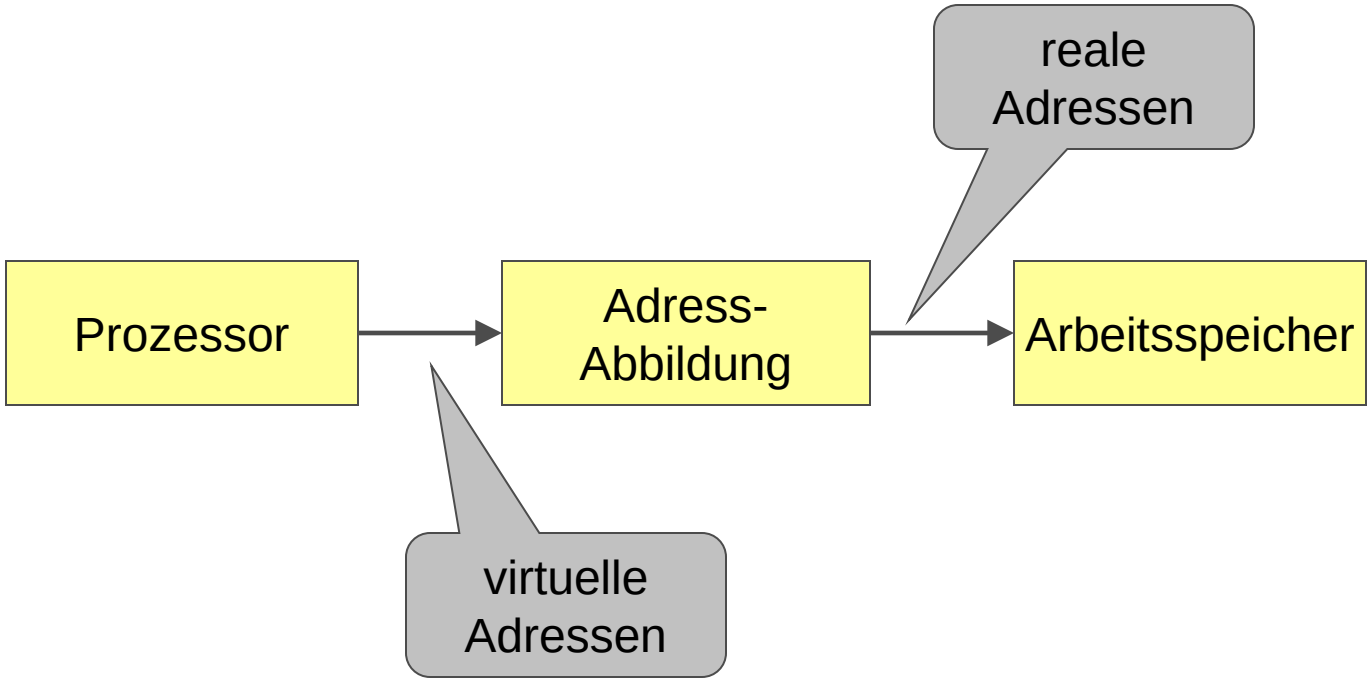
- Geräte werden einheitlich als Block von Speicherzellen behandelt.
  - Gerät wird über einen festgelegten Adressraum definiert.
  - Daten können in größeren Datenblöcken übertragen werden.

- Bewertung
  - Prozessorintegrierte Ein-/Ausgabe erfordert Unterstützung der Geräte bis zur Maschine M_0 durch alle Schichten hindurch.
  - Speicherintegrierte Ein-/Ausgabe erlaubt die einheitliche Integration beliebiger Geräte.

Arbeitsspeicher

- fast ausnahmslos heute Virtualisierung des Speichers
  - Prozessor bildet einen virtuellen Speicherbereich (z.B. 4 oder 64 GB) auf einen physikalischen, seitenweise strukturierten Bereich ab.
  - Seitenverwaltung erfolgt durch die Memory Management Unit (MMU)
- Speicher kann mit allgemeinen Strategien verwaltet werden.
- Mehrere virtuelle Speicher können parallel existieren.

Arbeitsspeicher: Abbildung von virtuellen Adressen



Übung

- Entwickeln Sie in Java eine Architektur für abstrakte Prozessoren. Prozessoren sollen folgende Funktionalität besitzen:
  - Prozessoren implementieren das Prozessor-Zustandsmodell.
  - Prozessoren besitzen UV, UR und NR.
  - Prozessoren können Unterbrechungen annehmen.

- Entwickeln Sie eine Methode, die das Rechnen des Prozessors simuliert. Diese Methode soll zufällig eine Unterbrechung auslösen, dabei Zustand und Registersatz ändern und sich nach einer zufällig gewählten Zeit beenden.

Selbstkontrolle

1. Warum genügt die Betrachtung einer abstrakten Prozessorarchitektur bei der Entwicklung eines Betriebssystems?
2. Welche Schichten müssen Sie bei der Migration auf eine neue Prozessorarchitektur anpassen?
3. Wie findet das Betriebssystem bei einer Unterbrechung das dazu gehörige Programm?
4. Warum sollte während der Behandlung von Unterbrechungen keine weitere Unterbrechung angenommen werden?
5. Skizzieren Sie das Modell der Prozessorzustände und Übergänge.
6. Warum ist die Verwendung virtueller Speicheradressen sinnvoll?

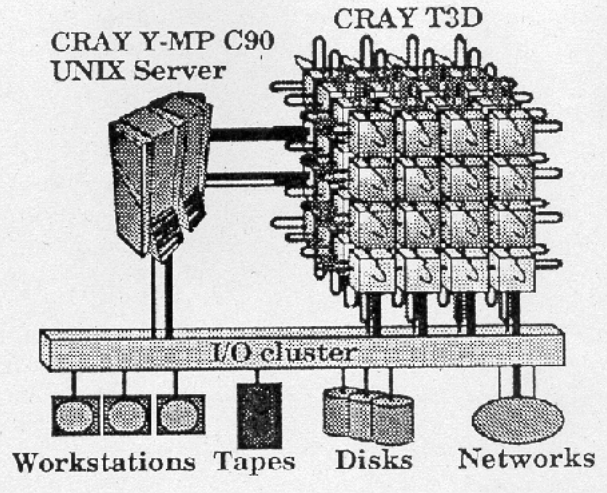
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Betriebssysteme
Prozesse



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Warum Prozesse?

- Betriebssystem muss die effiziente Benutzung von Betriebsmitteln gewährleisten.
  - einige Aufgaben können parallel bearbeitet werden
- Parallele Verarbeitung von Aufgaben muss strukturiert werden
  - Zur Beschreibung wird das Abstraktionsmittel Prozess eingeführt.

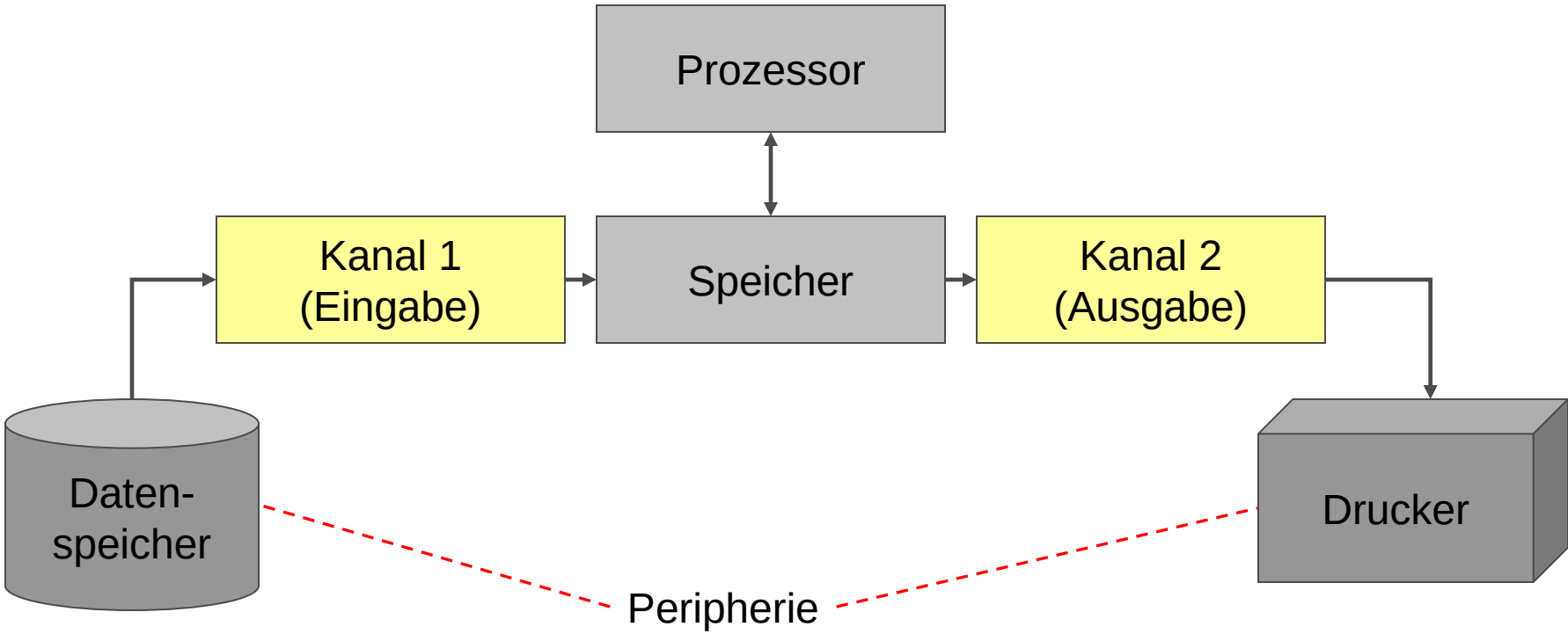
Beispiel für Prozessbearbeitung

- System zur Bearbeitung von Benutzer-Aufträgen (Jobs)

- Schritte in einem Job (frei definiert)
  1. Job lesen
  2. Job bearbeiten (übersetzen und ausführen)
  3. Ergebnis ausgeben

- durchschnittliche Bearbeitungszeiten (frei definiert)
  - Lesen: 0,3 sek.
  - Bearbeiten: 0,5 sek.
  - Ausgeben: 0,6 sek.

Beispiel für Prozessbearbeitung



Sequenzielle Organisation von Prozessen

- Das Betriebssystem bearbeitet alle Schritte streng sequentiell.

- Arbeitsweise als Pseudo-Code:

```
LOOP
        Lesen;
        Warte auf Ende des Lesens;
        Übersetzen;
        Ausführen;
        Drucken;
        Warte auf Ende des Druckens;
END
```

Bewertung der Leistungsfähigkeit

- Kennzeichen zur Beurteilung der Leistung eines Betriebssystems sind
  - der Durchsatz (d) von Jobs durch den Rechner
  - die jeweilige Auslastung (a) von Prozessor und Peripherie

Definition: Durchsatz

- Der Durchsatz eines Betriebssystems ergibt sich aus der mittleren Zahl von Jobs, die in einer bestimmten Zeiteinheit verarbeitet werden können.

$$d = \frac{Anzahl\_Jobs}{Zeiteinheit}$$

Definition: Auslastung

- Die Auslastung (a) einer Betriebssystem-Komponente ergibt sich aus der mittleren Belegungszeit der Komponente pro Job und der mittleren Bearbeitungszeit für einen Job.

$$a = \frac{mittlere\_Belegungszeit\_der\_Komponente\_pro\_Job}{mittlere\_Bearbeitungszeit\_eines\_Jobs}$$

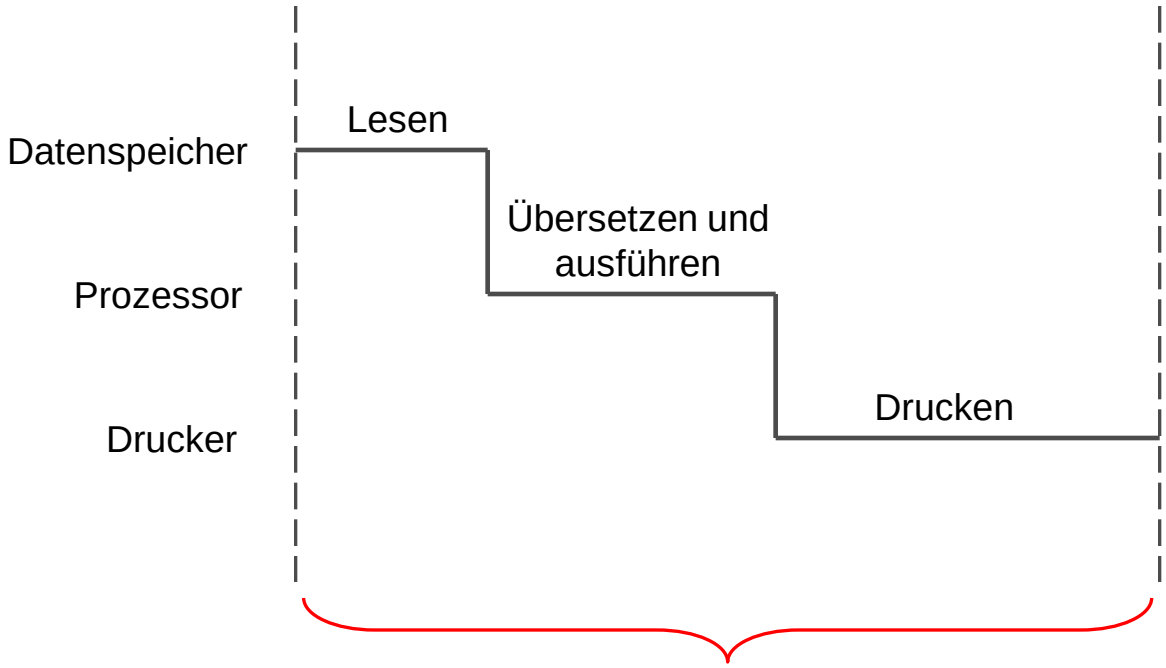
Auslastung und Durchsatz aus dem Beispiel

- Durchsatz d
  - pro Job: 0,3 sek. + 0,5 sek. + 0,6 sek. = 1,4 sek.
  - Durchsatz d = 60/1,4 = 43 Jobs pro Minute

- Auslastung a
  - Leseeinheit: 0,3 sek. / 1,4 sek. = 21 %
  - Prozessor: 0,5 sek. / 1,4 sek. = 36%
  - Drucker: 0,6 sek. / 1,4 sek. = 43%

- Bewertung
  - Prozessor muss die meiste Zeit untätig warten, bis die Peripherie fertig ist.
  - Rechenleistung wird unzureichend genutzt.

Auslastung bei sequenzieller Verarbeitung

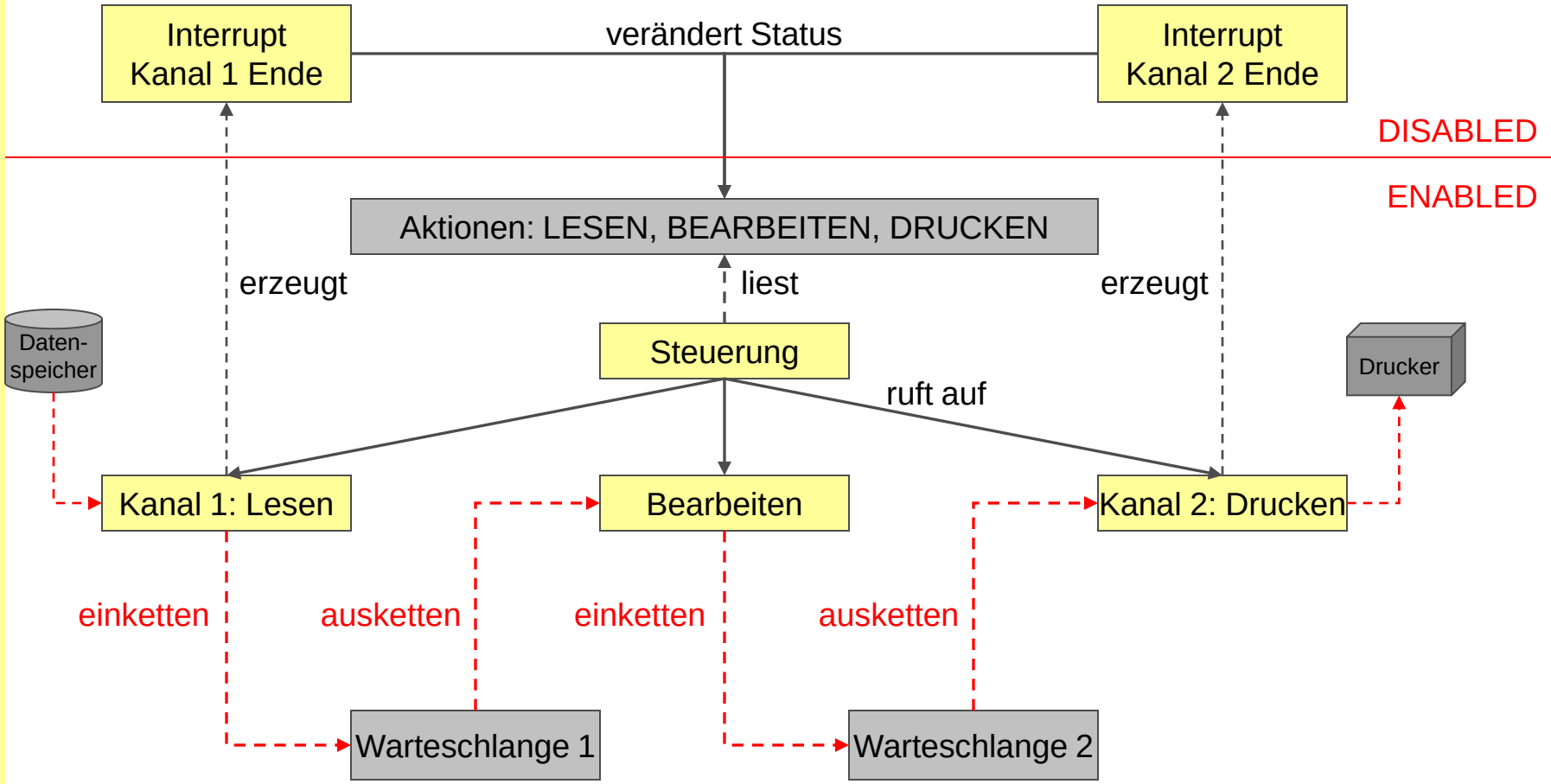


Organisation mit Unterbrechungen

- Auslastung und Durchsatz können dadurch erhöht werden, dass alle Komponenten des Rechners möglichst ständig in Betrieb sind.

- Ansatz zur Optimierung
  - definiere eine Aktionsliste, die für jeden Verarbeitungsschritt notiert, ob Aufträge vorliegen
  - erzeuge Unterbrechungen, wenn ein Schritt bearbeitet ist

Organisation mit Unterbrechungen



Implementierung: Speicher, Steuerung und Unterbrechungsbehandlung

```
Job[] Warteschlange1, Warteschlange2;
boolean LESEN      =false,
        BEARBEITEN =false,
        DRUCKEN    =false;

Interrupt_Kanal1_Ende () { DRUCKEN = false; }
Interrupt_Kanal2_Ende () { LESEN = false; }

Warten () { Null-Operation; }

Steuerung () {
  while (true) {
    if (DRUCKEN) then Drucken();
    else if (LESEN) then Lesen();
    else if (BEARBEITEN) then Bearbeiten();
    else Warten();
  }
}
```

Implementierung: Verarbeitungsschritte

```
Lesen () {
  LESEN = false;
  if (Job in Datenspeicher) {
    Job von Datenspeicher lesen;
    Job in Warteschlange1 einketten;
    BEARBEITEN = true;
  }
}

Drucken () {
  DRUCKEN = false;
  if (Job in Warteschlange2) {
    Job aus Warteschlange2 ausketten;
    Job drucken;
  }
}
```

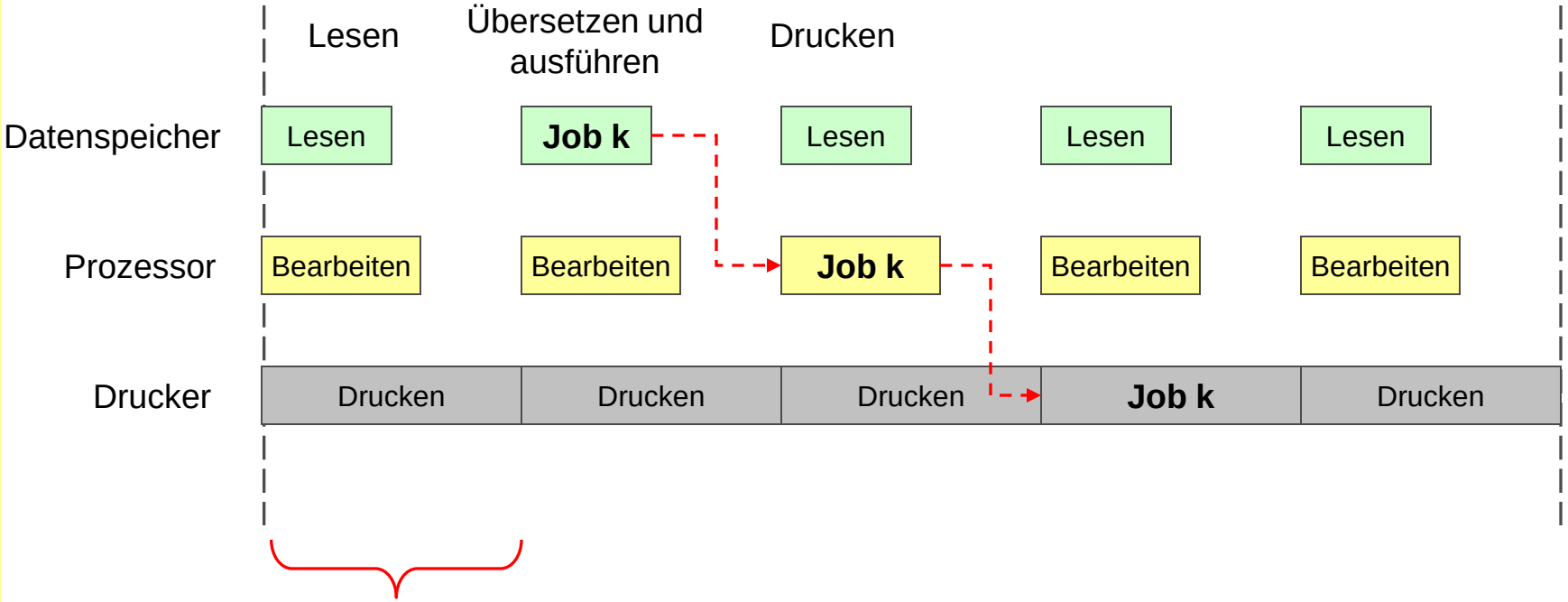
Implementierung: Verarbeitungsschritte

```
Bearbeiten () {
  Kette Job aus Warteschlange1;
  Job übersetzen;
  Job ausführen;
  Job in Warteschlange2 einketten;
  DRUCKEN = true;
  if (Warteschlange1 ist leer) BEARBEITEN = false;
}
```

Bewertung mit optimierten Bedingungen

- optimierende Annahmen
  - Warteschlangen laufen nicht über
  - Registersatz NR für das Steuerprogramm initialisiert
  - Prozessorbedarf für verwaltende Vorgänge wird vernachlässigt
- Durchsatz d
  - richtet sich nach dem aufwändigsten Teilschritt
  - d = 1 / 0,6 sek. = 100 Jobs pro Minute
- Auslastung a
  - während des Druckens können parallel Lesen und Bearbeiten erfolgen.
  - Prozessor: 0,5 sek. / 0,6 sek. = 83%
  - Eingabe: 0,3 sek. / 0,6 sek. = 50%
  - Drucker: 0,6 sek. / 0,6 sek. = 100%

Auslastung bei unterbrechungs-gesteuerter Verarbeitung



mittlere Bearbeitungszeit für 1 Job

Parallele Organisation

- Verarbeitungsschritte könnten durch Warteschlangen entkoppelt parallel zueinander arbeiten.

- Ansatz zur Optimierung
  - trenne Verarbeitungsschritte in parallel ablaufende Module
  - Module laufen unabhängig ab

- Pseudoprozessor als Abstraktion für Parallelität
  - Programm bzw. Modul erhält einen Pseudoprozessor
  - Pseudoprozessoren implementieren die Verarbeitungsschritte
  - unterliegende Maschine übernimmt die Wechsel in der Bearbeitung

Implementierung: Speicher und Steuerung

```
Job[] Warteschlange1, Warteschlange2;

Steuerung () {
  PARALLEL {
    Drucken();
    Lesen();
    Bearbeiten();
  }
}
```

Implementierung: Verarbeitungsschritte

```
Lesen () {
  while (true) {
    if (Job in Datenspeicher) {
      Job von Datenspeicher lesen;
      Job in Warteschlange1 einketten;
    }
  }
}

Drucken () {
  while (true) {
    if (Job in Warteschlange2) {
      Job aus Warteschlange2 ausketten;
      Job drucken;
    }
  }
}
```

Implementierung: Verarbeitungsschritte

```
Bearbeiten () {
  while (true) {
    if (Job in Warteschlange 1) {
      Kette Job aus Warteschlange1;
      Job übersetzen;
      Job ausführen;
      Job in Warteschlange2 einketten;
    }
  }
}
```

Threads = Pseudoprozessoren

- Prozesse werden mit Hilfe von Pseudoprozessoren realisiert.

- Bestandteile eines solchen Prozesses
  - Programm
  - Pseudoprozessor
  - Laufzeitkeller für temporäre Daten

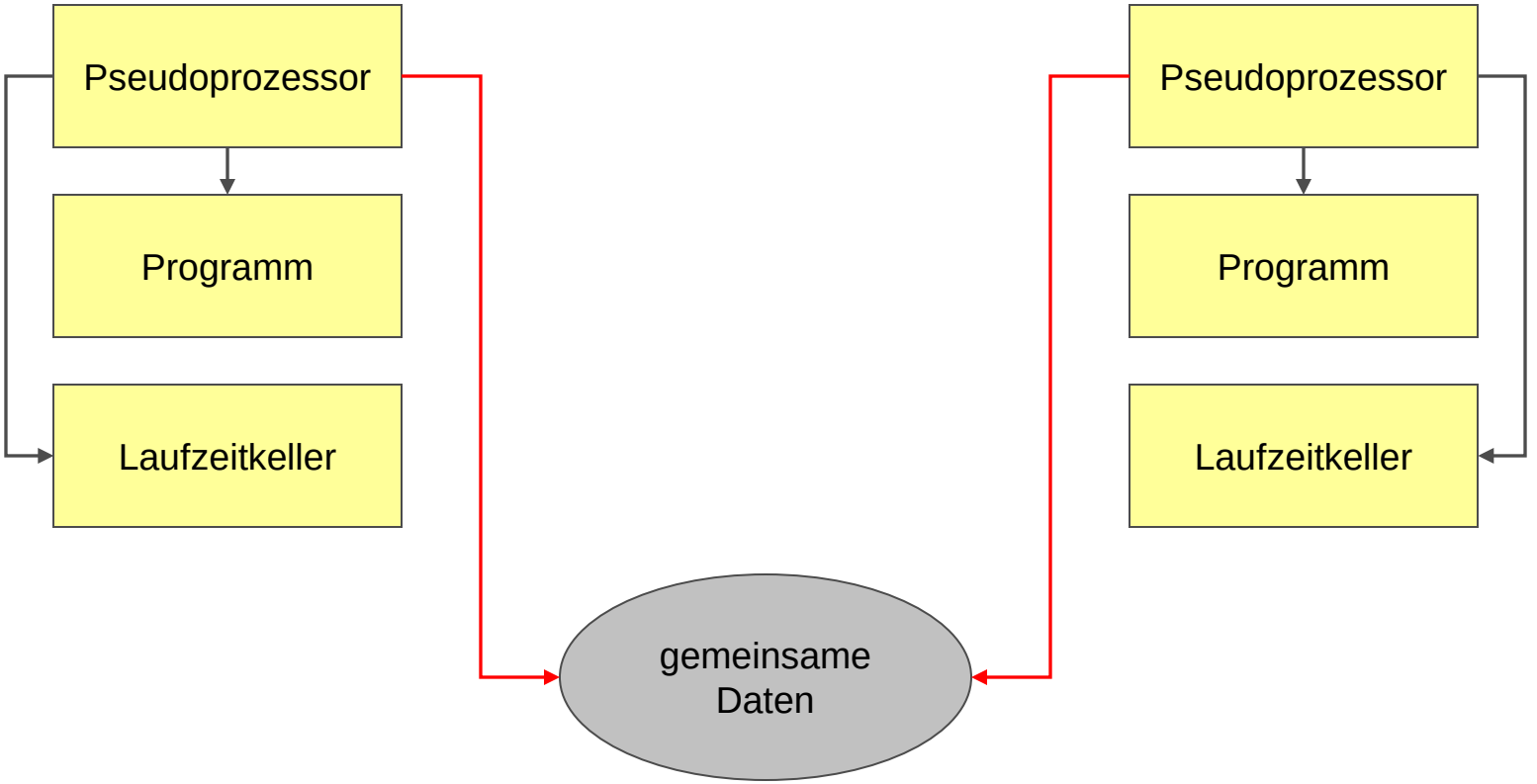
- prozess-spezifischer Speicher
  - Programm
  - Laufzeitkeller
  - nicht-lokale Speicherbereiche

Speicherbereiche von Threads

- Programm ist ablaufinvariant: reentrant code
  - Mehrere Prozesse des gleichen Typs können parallel gestartet werden
    - Die Prozesse haben unterschiedliche Bearbeitungszustände.
  - Prozesse überlappen bei der Verwendung nicht-lokaler Daten
    - z.B. Warteschlangen des letzten Beispiels

- Realisierung von Prozessen im Betriebssystem
  - Speicherung von Verwaltungsinformationen
  - Es müssen verwaltende, speicher-residente Prozesse bei Start des Betriebssystems geladen werden
    - z.B. Prozess ID 1 bei Unix/Linux

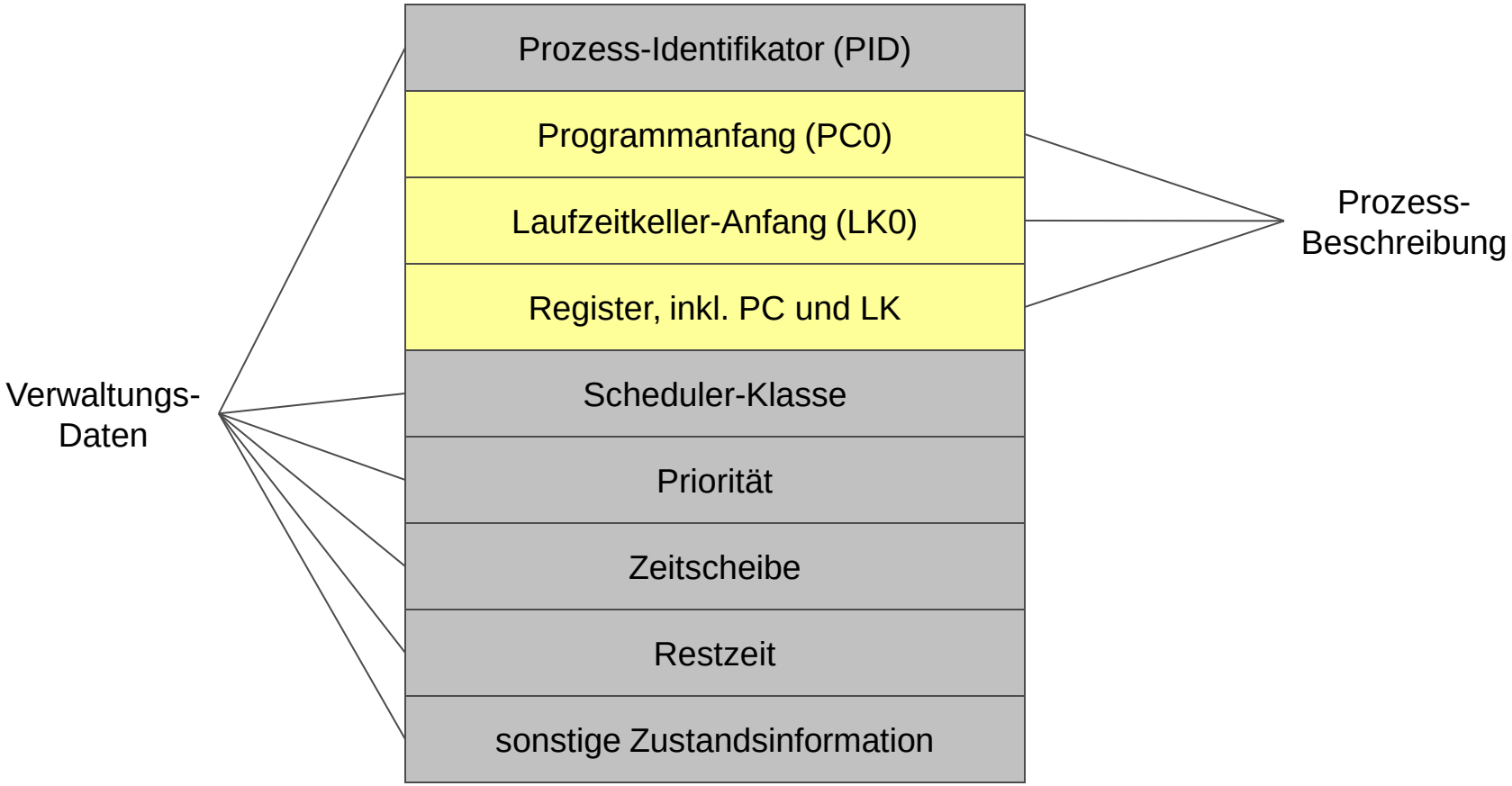
Organisation von Prozessen



Threads für Monoprozessor-Systeme

- Threads besitzen eine Datenstruktur, die den Zustand eines realen Prozessors beschreiben.
- Zusätzliche Informationen: Daten zur Steuerung von Prozessumschaltungen (Scheduling)
  - z.B. Priorität, bisherige Rechenzeit, geschätzter Restaufwand
- Threads werden mit einem Process Control Block (PCB) beschreiben.

Process Control Block (PCB)



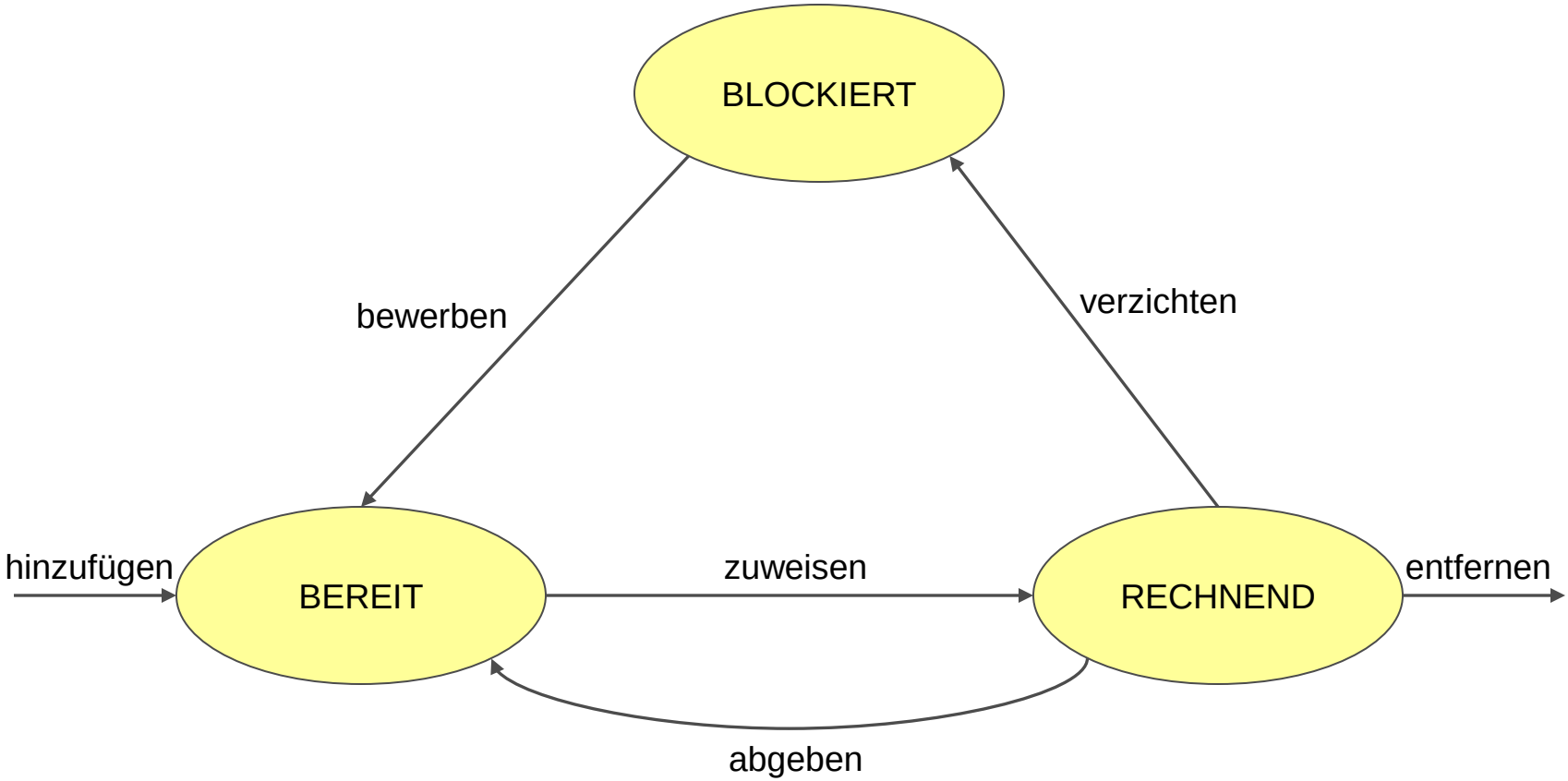
Umschalten von Pseudoprozessoren

- Ablaufplanung, engl. scheduling
  - realer Prozessor wählt aus ablaufbereiten Prozessen einen geeigneten aus
  - ausgewählter Prozess läuft für eine definierte Zeit
  - Ereignisse (z.B. Interrupts) erwirken die Umschaltung zwischen Prozessen
    - auch Starten oder Beenden
    - Umschaltung kann auch freiwillig (vom Prozess ausgelöst) erfolgen, z.B. durch sleep-Anweisung
  - wird auch als Prozessor-Multiplexen bezeichnet
  - Prozesse werden mit einem Zustand versehen

Prozess-Zustandsmodell

- Prozessor-Mulitplex-Mechanismus weist Prozessen 3 mögliche Zustände zu
  - RECHNEND
    - ist ein Prozess, dem ein realer Prozessor zugeteilt ist
  - BEREIT
    - ist ein Prozess, der sich um die Zuteilung eines Prozessors bewirbt
  - BLOCKIERT
    - ist ein Prozess, der auf die Zuteilung eines realen Prozessors verzichtet

Prozess-Zustandsmodell



Übung

- Entwickeln Sie in Java eine Architektur für Prozesse. Prozesse sollten dabei folgende Funktionalität aufweisen:
  - Prozesse besitzen einen PCB.
  - Prozesse implementieren das Prozess-Zustandsmodell.

- Entwickeln Sie eine Architektur zur Verwaltung von Prozessen eines Betriebssystems.
  - Die Verwaltung speichert eine oder mehrere Listen von Prozessen.
  - Die Verwaltung speichert eine oder mehrere Listen von Prozessoren.

Implementierung des Zustandsmodells

- Das Zustandsmodell wird als Modul einer abstrakten Maschine implementiert.
  - Maschine wird als Nukleus bezeichnet.

- Dispatcher = Modul des Zustandsmodells
  - verwaltet getrennte Listen von Prozessen in den Zuständen BLOCKIERT, BEREIT, RECHNEND
  - PCBs speichern einen (oder mehrere) Zeiger für die Listenverwaltung
    - ein Zeiger: lineare Liste
    - zwei Zeiger: beidseitig verkettete Liste

Null-Prozess

- Dispatcherimplementierungen fordern, dass immer ein Prozess BEREIT ist.
  - Prozessoren besitzen eine Null-Operation
- Betriebssysteme implementieren einen Null-Prozess
  - Prozessor tritt nie in einen Wartezustand
  - bewirbt sich ständig neu um die Zuweisung eines realen Prozessors
  - z.B. Leerlaufprozess unter Windows

Implementierung des Nukleus

- Forderungen an eine Implementierung
  - nach Verlassen muss sich für jeden Prozessor genau ein Prozess im Zustand RECHNEND befinden
  - Initialisierung des Nukleus erzeugt den ersten Prozess
    - Vaterprozess aller Prozesse im Betriebssystem

- Muster für die Implementierung des Nukelus
  1. keine Steuerung des Dispatchers
  2. Einketten eines neuen oder bewerbenden Prozesses
  3. Abgeben, Verzichten oder Entfernen eines rechnenden Prozesses, dann neue Zuweisung des realen Prozessors
  4. Kombination aus 2 und 3.

Implementierung des Nukleus

- Aktivierung des Nukleus
  - Prozesswechsel im Nukleus darf nicht durch Interrupts unterbrochen werden
  - Zugang zum Nukelus erfolgt durch softwareseitige Unterbrechung (Traps)

- Beenden eines Prozesses
  - wird ein Prozess beendet, muss er sein Entfernen aus dem Nukleus anstoßen
  - Entfernen muss auch bei zwangsweisem Beenden erfolgen

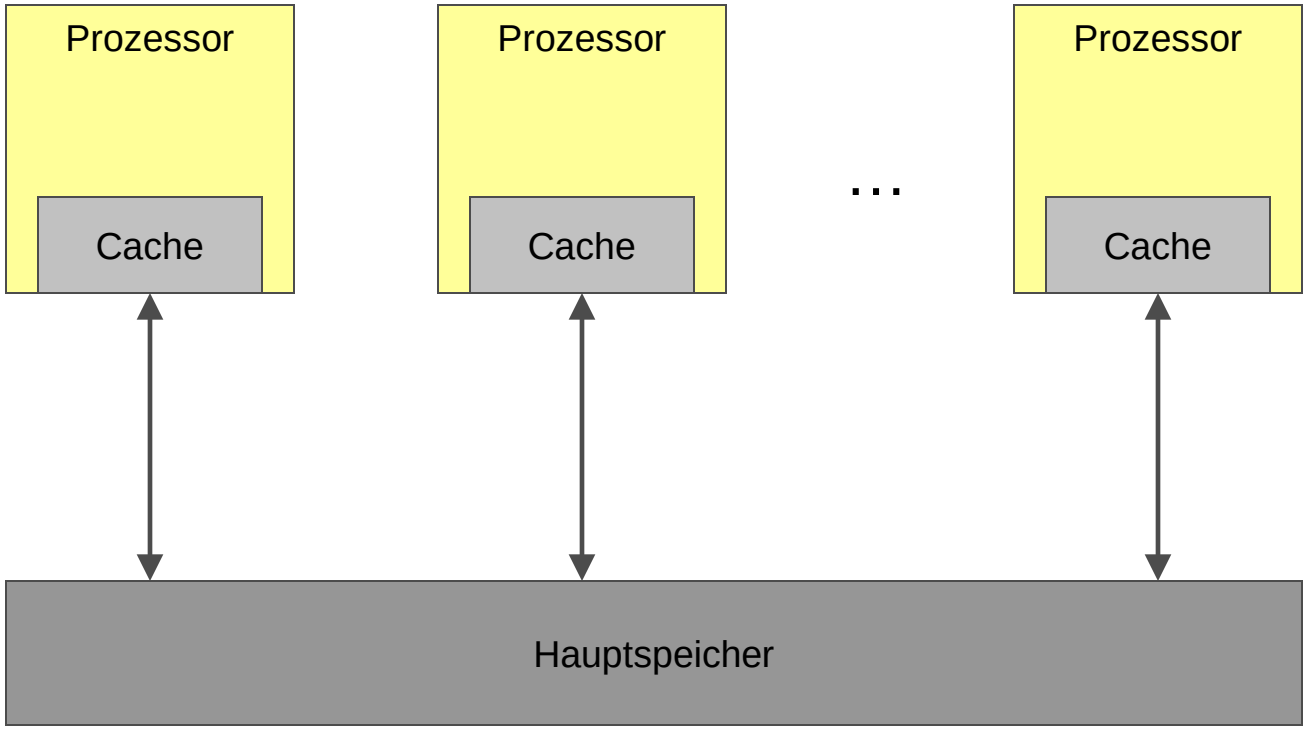
Threads bei symmetrischen Mehrprozessorsystemen

- symmetrisches Mehrprozessorsystem
  - mehrere gleiche Prozessoren mit gemeinsamem Arbeitsspeicher
  - jeder Prozessor kann Aufgaben eines anderen Prozessors übernehmen
- Änderungen im Betriebssystem
  - Unterbrechungen müssen an den passenden Prozessor geleitet werden.
  - Bei n Prozessoren müssen sich genau n Prozesse im Zustand RECHNEND befinden

Prozessoren mit Cache

- Was ist ein Cache?
  - sehr schneller, prozessorlokaler Speicher, der einen Teil des Arbeitsspeicher für einen schnelleren Zugriff aufnimmt.
- Warum gibt es Caches?
  - Prozesse besitzen ein Lokalitätsverhalten
    - Speicherzellen, die referenziert worden sind, werden in einem engen Zeitfenstern noch mindestens einmal referenziert
    - Zugriffe auf verschiedene Zellen liegen räumlich dicht beieinander.
- Aufgaben des Betriebssystems
  - Speicher (-Block) für Zugriffe rechtzeitig im Cache bereitstellen
  - Cache und Hauptspeicher konsistent halten
    - Cache-Kohärenz-Protokolle

Multiprozessorsystem mit Caches



Organisation des Caches

Gültig-Bit	Verwaltungs-daten	Adresse im Hauptspeicher	Speicherblock
Gültig-Bit	Verwaltungs-daten	Adresse im Hauptspeicher	Speicherblock
Gültig-Bit	Verwaltungs-daten	Adresse im Hauptspeicher	Speicherblock

.
.
.
.

Gültig-Bit	Verwaltungs-daten	Adresse im Hauptspeicher	Speicherblock

Cache-Kohärenz-Protokolle: Begriffe

- read-hit
  - Ein lesender Zugriff findet den referenzierten Block im Cache vor.
- read-miss
  - Ein lesender Zugriff findet den referenzierten Block nicht im Cache vor.
- write-hit
  - Ein schreibender Zugriff findet den referenzierten Block im Cache vor.
- write-miss
  - Ein schreibender Zugriff findet den referenzierten Block nicht im Cache vor.

Cache-Kohärenz-Protokolle: Write Through

- Strategie
  - Schreibzugriffe führen immer zum Durchschreiben des Blocks in den Hauptspeicher.

- Bewertung
  - einfach, aber ineffizient
  - Änderungen sind sofort im Hauptspeicher sichtbar
  - Prozessorumschaltung erfordert keinerlei zusätzliche Maßnahmen

Cache-Kohärenz-Protokolle: Write Through

- read-hit
  1. Lesen des referenzierten Blocks aus dem Cache.
- read-miss
  1. Transport des betreffenden Blocks vom Hauptspeicher in den Cache.
  2. Lesen des Blocks aus dem Cache.
- write-hit
  1. Invalidierung aller eventuell vorhandenen Kopien des Blocks in benachbarten Caches.
  2. Durchschreiben des Blocks in den Hauptspeicher.
- write-miss
  1. Invalidierung aller eventuell vorhandenen Kopien des Blocks in benachbarten Caches.
  2. Durchschreiben des Blocks in den Hauptspeicher.
  3. Transport des betreffenden Blocks vom Hauptspeicher in den Cache.

Cache-Kohärenz-Protokolle: Copy-Back

- Strategie
  - Schreibzugriffe im lokalen Cache zulassen, solange kein Konflikt in Form konkurrierender Lese- oder Schreibzugriffe durch andere Prozessoren vorliegt.

- Bewertung
  - Rückschreiben in den Hauptspeicher wird minimiert.
  - Hauptspeicher kann bei Prozessorumschaltung inkonsistent sein.
    - Vor Umschaltung müssen alle sog. DIRTY Blöcke in den Hauptspeicher zurückkopiert werden.
      - Prozessoren besitzen dafür eine Instruktion FLUSH.

Cache-Kohärenz-Protokolle: Copy-Back

- read-hit
  1. Lesen der referenzierten Speicherzelle aus dem Cache.
- read-miss
  1. Block liegt im Zustand DIRTY in externem Cache vor.
     a. Zustand DIRTY → READ in externem Cache.
     b. Rückschreiben des Blocks in den Hauptspeicher.
     c. Block in lokalen Cache (Zustand READ) lesen.
     d. Block aus Cache lesen.
  2. Block liegt nicht im Zustand DIRTY in externem Cache vor.
     a. Block vom Hauptspeicher (Zustand READ) lesen.
     b. Block aus Cache lesen.

Cache-Kohärenz-Protokolle: Copy-Back

- write-hit
  1. Block im Zustand DIRTY
     a. Schreibzugriff in eigenem Cache ausführen.
  2. Block im Zustand READ
     a. Block in allen Nachbar-Caches invalidieren.
     b. Zustand → DIRTY
     c. Schreibzugriff in eigenem Cache durchführen.
- write-miss
  1. Wenn Block im Zustand DIRTY in externem Cache: Rückschreiben des Blocks in den Hauptspeicher veranlassen.
  2. Block in allen Nachbar-Caches invalidieren.
  3. Block vom Hauptspeicher in den Cache kopieren.
  4. Zustand → DIRTY
  5. Schreibzugriff in eigenem Cache durchführen.

Cache-Überlauf

- Bei Überlauf des Caches müssen „alte“ Blöcke verdrängt werden.
  - nicht DIRTY: löschen
  - DIRTY: zurückschreiben
- Betrachtete Cache-Kohärenz-Protokolle berücksichtigen Überläufe nicht.

Prozessor-Scheduling

- Scheduling ist eine Auswahlstrategie, nach der ein knappes Betriebsmittel im Wettbewerb befindlichen Prozessen zugewiesen wird.
  - Scheduler = die Auswahlstrategie realisierendes Programm

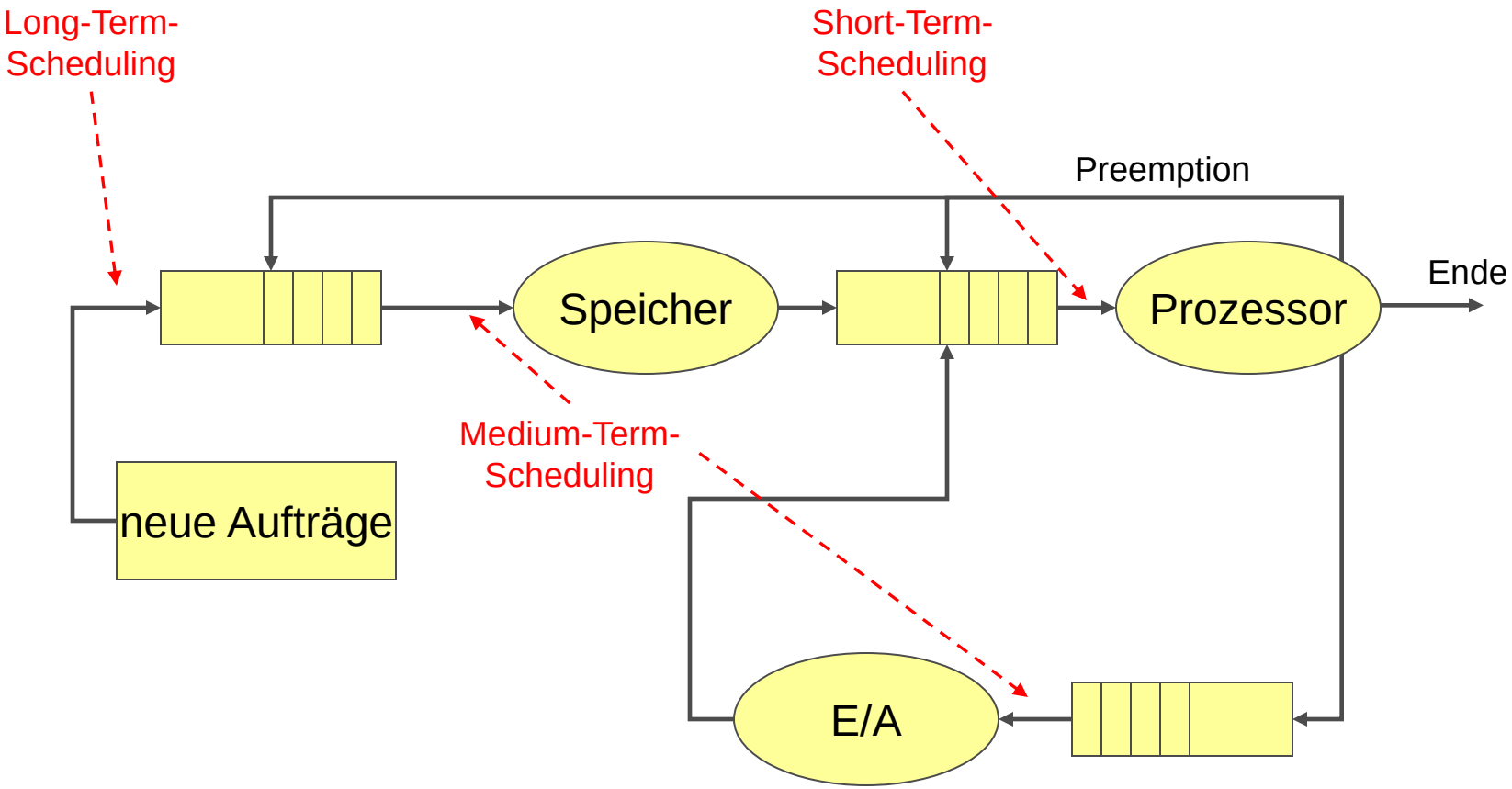
- Schedulingstrategien unterscheiden sich durch den Planunungszeitraum
  - Long-Term-Scheduling
  - Medium-Term-Scheduling
  - Short-Term-Scheduling

Betriebssystem aus Sicht des Schedulers

- Ein Betriebssystem kann als Warteschlangensystem aufgefasst werden, in dem sich Aufträge um die Zuteilung von Betriebsmitteln bewerben.

- Einsatz mehrerer Scheduling-Strategien
  - Long-Term-Scheduler kontrollieren die Wartschlange von Stapelverarbeitungs-Aufträgen (engl. batch jobs).
  - Medium-Term-Scheduler kontrollieren die Zuteilung von Arbeitsspeicher und Ein-/Ausgabegeräten.
  - Short-Term-Scheduler kontrollieren die Prozessor-Zuteilung.

Betriebssystem aus Sicht des Schedulers



Ziel des Schedulings

- Scheduler optimieren ein vorgegebenes Systemverhalten. Typische Kriterien:
  - Maximierung der Prozessor/Speicher Auslastung
  - Maximierung des Durchsatzes (Anzahl Aufträge/Zeit)
  - Minimierung der Durchlaufzeit (engl. turnaround time)
    - Summe der Bedien- und Wartezeiten
  - Minimierung der Antwortzeit (engl. response time)
    - Zeit von der Anforderung bis zur Bedienung
- Wichtigste Eigenschaft: Fairness
  - Konkurrierende Prozesse erhalten des Betriebsmittel in einer endlichen Zeit zugeteilt.
  - Unendlich lange Wartestellungen sind ausgeschlossen.

Eigenschaften von Scheduling-Disziplinen

- Unterbrechbarkeit von Prozessen
  - preemptiv: einem Prozess können Betriebsmittel entzogen werden
  - nicht-preemptiv: zugeteilte Betriebsmittel verbleiben bis zur freiwilligen Abgabe durch den Prozess

- Hilfsmittel bei der Steuerung
  - Timeouts: Begrenzung der Rechenzeit
  - Prioritäten: Festlegung der Wichtigkeit der Bearbeitung

Scheduling Disziplin: First-Come-First-Served (FCFS)

- Prozessor wird in der zeitlichen Reihenfolge der Anforderungen zugeteilt.

- Bewertung
  - nicht preemptiv
  - extreme Benachteiligung von Kurzläufern
  - ungeeignet für einen Dialogbetrieb

Beispiel: FCFS

- 3 Prozesse mit vorgegebenen Laufzeiten (Zeiteinheit beliebig):
  - Prozess 1 = 24
  - Prozess 2 = 3
  - Prozess 3 = 3
  - Reihenfolge: 1,2,3

- mittlere Durchlaufzeit: d = (24+27+30)/3 = 27

Prozess 1 (24)	Proz. 2 (3)	Proz. 3 (3)

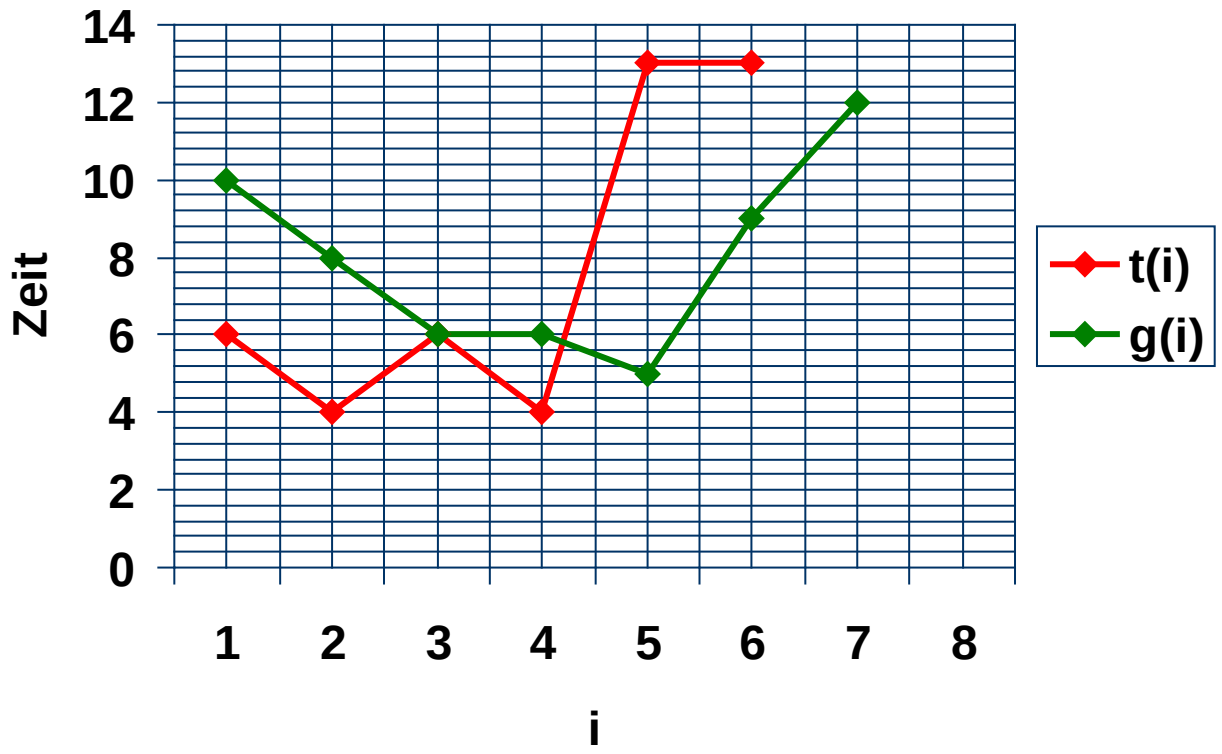
Scheduling Disziplin: Shortest Job First (SJF)

- Prozesse mit kurzer Bedienzeit werden bevorzugt.
  - Prozesse werden sortiert nach Zeitbedarf bis zur nächsten freiwilligen Abgabe, sog. processor burst
- Problem
  - kommender Zeitbedarf muss vor der Sortierung bekannt sein: in der Praxis unmöglich!
- Ansatz
  - processor burst schätzen aus letzter geschätzter Zeit und letzter gemessener Zeit
    - g_i: i-te Zeitschätzung
    - t_i: i-te Zeitmessung
    - a: Gewichtungsfaktor [0..1]
  - Schätzung des nächsten processor burst
    - $g_{i+1} = a \bullet t_i + (1-a) \bullet g_i$

Scheduling Disziplin: Shortest Job First (SJF)

- Bewertung
  - nicht preemptiv
  - theoretisch die optimale Strategie
  - länger laufende Prozesse mit langen processor bursts werden extrem benachteiligt
  - Schätzungen liefern gleichen Zeitbedarf bei allen Prozessen: FCFS

Beispiel: SJF, Schätzung des nächsten processor bursts, a=0,5



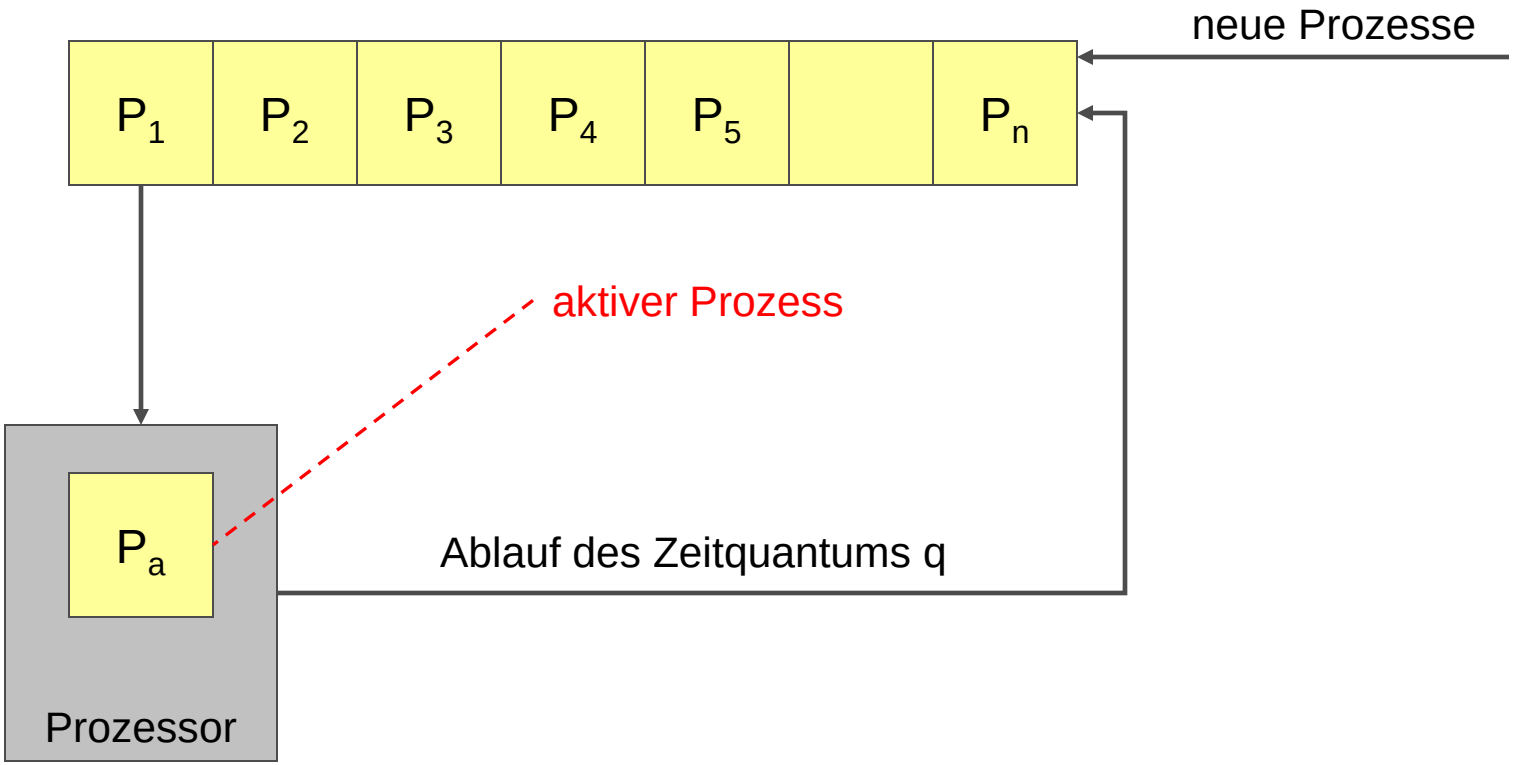
Scheduling Disziplin: Round Robin (RR)

- Kurzläufer werden bei unbekannten Laufzeiten bevorzugt.
  - Prozesse werden in der zeitlichen Reihenfolge ihres Eintreffens in eine Warteschlange eingefügt.
  - Zur Bearbeitung eines Prozesses steht ein Zeitquantum q (auch Zeitscheibe) zur Verfügung.
    - Gibt der Prozess innerhalb von q nicht freiwillig ab, wird er unterbrochen.
  - Unterbrochene Prozesse werden wieder ans Ende der Warteschlange eingefügt.

Scheduling Disziplin: Round Robin (RR)

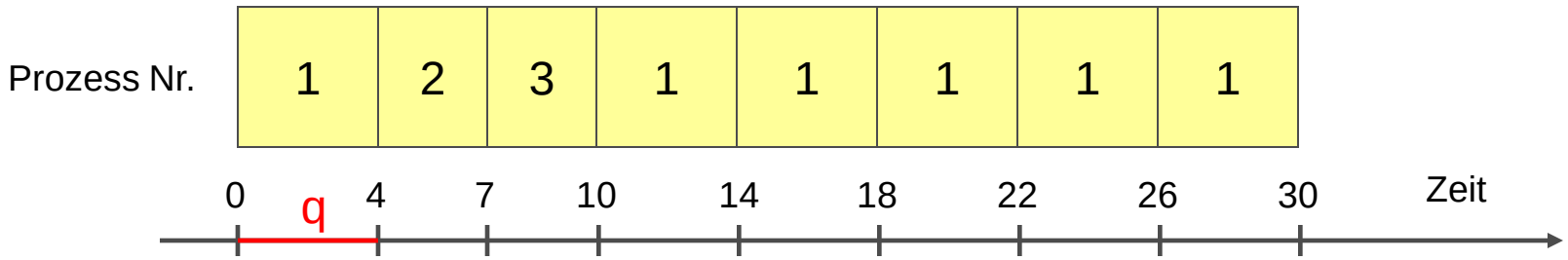
- Bewertung
  - preemptiv
  - für den Dialogbetrieb gut geeignet
  - Durchlaufzeiten sind gegenüber FCFS deutlich geringer

Scheduling Disziplin: Round Robin (RR)



Beispiel: Durchlaufzeiten bei RR

- 3 Prozesse mit vorgegebenen Laufzeiten (Zeiteinheit beliebig):
  - Prozess 1 = 24
  - Prozess 2 = 3
  - Prozess 3 = 3
  - Reihenfolge: 1,2,3
  - Zeitquantum q = 4



Beispiel: Durchlaufzeiten bei RR

- Durchlaufzeiten
  - Prozess 1 = 30
  - Prozess 2 = 7
  - Prozess 3 = 10

- mittlere Durchlaufzeit: (30+7+10) / 3 < 16

Spezialisierung von RR: Anpassung des Zeitquantums q

- Zeitquantum q wird lastabhängig festgelegt
  - n = Zahl der Prozesse
  - q_0 = initiales Zeitquantum
  - q_{min} = Untergrenze
  - $q = q_0 / n$

- Bewertung
  - $q < q_{min}$: spürbare Verlängerung der Antwortzeiten im Dialogbetrieb
  - $q \rightarrow 0$: alle Prozesse laufen quasi parallel, aber um den Faktor n verlangsamt
    - Wunschverhalten für Dialogbetrieb
  - $q \rightarrow \infty$: FCFS

Scheduling Disziplin: Feedback Scheduling (FS)

- Zusammenfassung mehrerer Scheduling-Disziplinen und Berücksichtigung der Vergangenheit der Prozesse.

- Beispiel rechenzeitabhängiges FS
  - Neuer Prozess wird mit hoher Priorität und niedrigen Zeitquantum ausgestattet.
  - Ist der Prozess nach Ablauf des Zeitquantums nicht fertig, so wird das Zeitquantum verdoppelt und die Priorität um eins reduziert.

- Bewertung
  - Prozesse mit hohem Rechenbedarf erhalten automatisch eine niedrige Priorität.
  - Kurzläufer können langlaufende Prozesse verdrängen.

Scheduling Disziplin: Multilevel Scheduling (MS)

- Prozesse werden in disjunkte Mengen geordnet und einem jeweils zuständigern Scheduler zugeordnet.
  - Bearbeitung der Prozessmengen definiert die Rangfolge der Prozesse.

- Beispiel: Unterscheidung zwischen System-, Dialog- und Hintergrundprozess
  - Systemprozess
    - Priorität = 1
    - Scheduler = First Come First Served FCFS
  - Dialogprozess
    - Priorität = 2
    - Scheduler = Round Robin (RR)
  - Hintergrundprozess
    - Priorität = 3 .. k
    - Scheduler = rechenzeitabhängiges Feedback Scheduling (FS)

Echtzeit Scheduling (RT), engl. realtime scheduling

- Rechnerinterne Aktionen müssen aufgrund eines Ereignisses innerhalb einer vorgegebenen Zeit abgeschlossen sein.
  - spätester Zeitpunkt = deadline
- Beispiel
  - spätestens 1 Sekunde nach Eintreffen des Ereignisses „Überdruck“ ein Ventil öffnen.
- Unterscheidung von Echtzeitsystemen
  - weich
    - Bei Überschreiten der Deadline entsteht ein fehlerhaftes Ergebnis: Ausschuss kann entstehen
  - hart
    - Bei Überschreiten der Deadline werden wichtige Funktionen beeinträchtigt: Lebens- bzw. Existenzgefahr

Scheduling Disziplin: Shortest Deadline First (SDF)

- Prozess mit der kürzesten Deadline erhält den Prozessor.
  - Verfahren setzt eine Systemuhr (Absolutzeit) voraus.
- Prozesse erhalten bei Einfügen in die Wartschlange im Process Control Block (PCB) der Zeitpunkt für das Erreichen der Deadline eingetragen:
  - $t_{deadline} = t_{aktuell} + t_{verbleibend}$
- Bewertung
  - preemptiv
  - Echtzeit Scheduling
  - alle Deadlines können eingehalten werden, solange ausreichend Rechenleistung zur Verfügung steht.
    - Problem wird auf die Dimensionierung von Hardware reduziert, was in der Praxis ein großes Problem darstellt.

Übung

- Erweitern Sie Ihre Prozess-Architektur um die Möglichkeit, einen Prozess zu unterbrechen und den Prozessor freizugeben.

- Entwickeln Sie eine Architektur für einen Scheduler. Implementieren Sie folgende Scheduling-Disziplinen:
  - First Come First Served (FCFS)
  - Round Robin (RR)
  - Feedback Scheduling (FS)

- Erweitern Sie Ihre Architektur der Prozessverwaltung um einen Scheduler. Der Scheduler soll austauschbar sein.

Übung

- Entwickeln Sie eine Laufumgebung mit mehreren Prozessen und simulieren Sie deren Berechnung.
  - Ermöglichen Sie möglichst viele Einstellmöglichkeiten Ihres Systems.
  - Protokollieren Sie die Aktionen Ihrer Umgebung in geeigneter Form in einer Textdatei.
  - Berechnen Sie Auslastung und Durchsatz Ihrer Laufumgebung.
- Erweitern Sie zu diesem Zweck ggfs. Ihre bisherigen Entwürfe.
- Vergleichen Sie die Scheduling-Disziplinen durch geeignete Testreihen.

Selbstkontrolle 1

1. Wie ist die Auslastung einer Systemkomponente definiert?
2. Wie ist der Durchsatz eines Systems definiert?
3. Erläutern Sie, warum eine unterbrechungsgesteuerte Auftragsverarbeitung einen höheren Durchsatz und eine höhere Auslastung erzeugt.
4. Worin liegt der Unterschied zwischen paralleler und unterbrechungsgesteuerter Verarbeitung?
5. Welche Bestandteile hat ein Process Control Block?
6. Beschreiben Sie das Prozess-Zustandsmodell: Skizze und Beschreibung der Zustände.
7. Warum kann ein Prozess nicht aus dem Zustand BEREIT entfernt werden?
8. In welcher Situation (ein Beispiel) ist ein Prozess im Zustand BLOCKIERT?

Selbstkontrolle 2

1. Was ist ein Cache?
2. Was versteht man unter Lokalitätsverhalten?
3. Erklären Sie Begriffe read-hit, read-miss, write-hit und write-miss.
4. Wie können Scheduling-Verfahren klassifiziert werden?
5. Welche Scheduler-Klasse eignet sich für welche Aufgaben des Betriebssystems?
6. Wie werden bei SJF neue Prozesse bedient, wenn die initiale Schätzung auf Null gesetzt wird?
7. Wann ist SJF äquivalent zu FCFS?
8. Warum eignet sich FCFS nicht für den Dialogbetrieb?

Selbstkontrolle 3

1. Welche Priorität sollte ein Null-Prozess besitzen?
2. Vergleichen Sie SJF mit SDF. Wo liegen die Unterschiede?

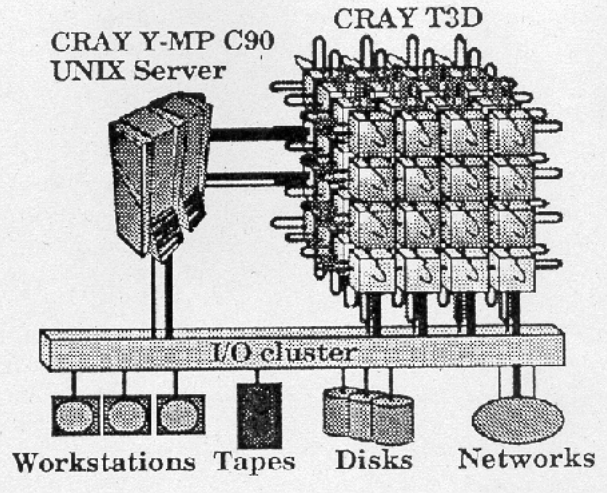
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Betriebssysteme
Prozess-Synchronisation



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

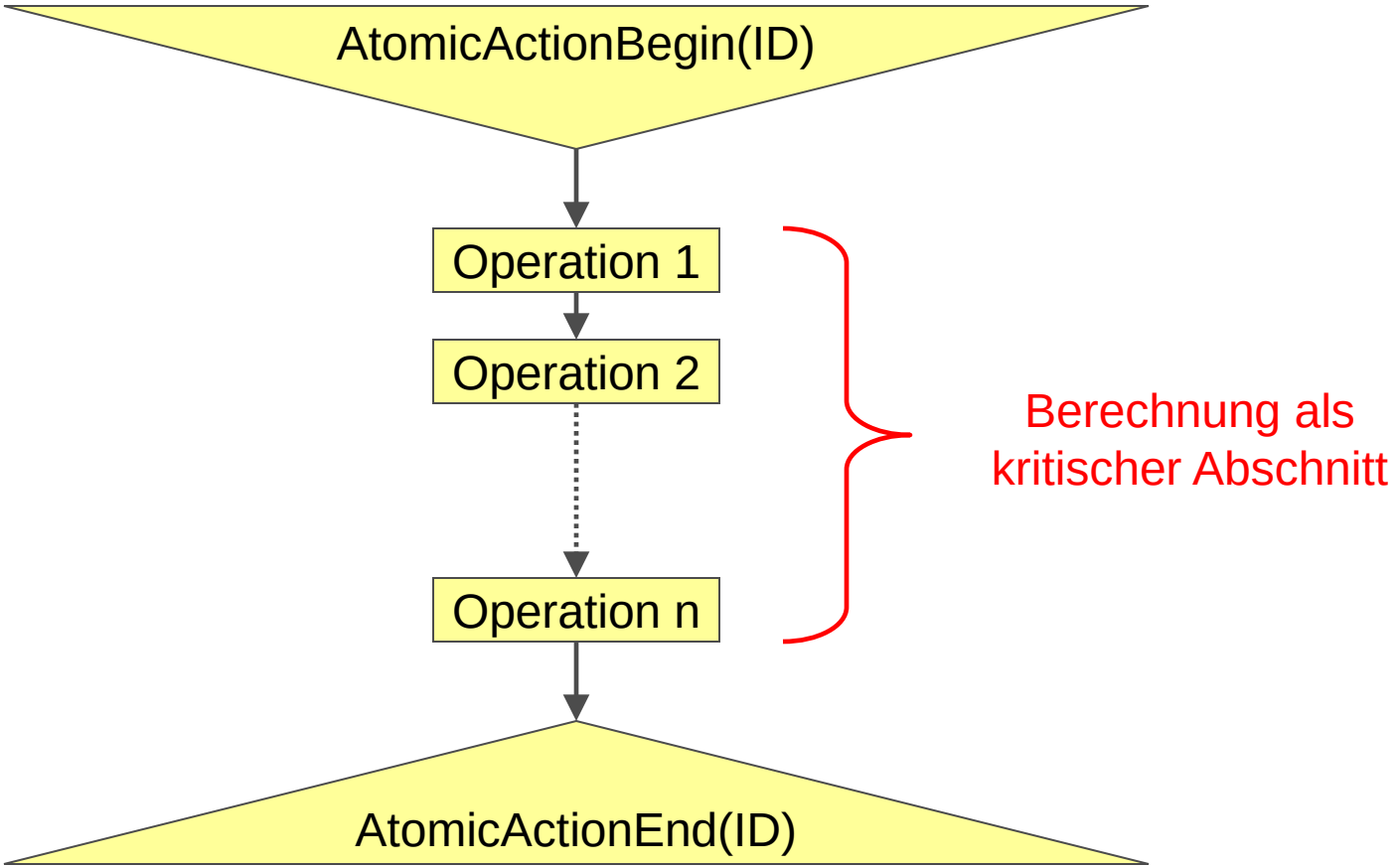
Verknüpfung von Prozessen

- Prozesse können miteinander agieren. Für eine konsistente Bearbeitung muss das Betriebssystem Mechanismen zur Synchronisation bereitstellen.
- Paradigmen für Prozess-Interaktion
  - Konkurrenz: Prozess-Synchronisation
    - Konkurrierende Prozesse befinden sich in einer Wettbewerbssituation, z.B. durch überlappenden Speicher. Zeitlich überlappende Berechnungen müssen so synchronisiert werden, dass eine (beliebige) Reihenfolge erzwungen wird. Konkurrierende Prozesse kennen sich üblicherweise nicht.
  - Kommunikation
    - Kommunizierende Prozesse tauschen gezielt Informationen aus, beteiligte Prozesse sind bekannt. Die beteiligten Prozesse müssen zeitlich synchronisiert werden.

Einfache Prozess-Synchronisation: Atomare Operationen

- Atomare Operation (engl. atomic action)
  - Jede Berechnung, die Daten mit externen, konkurrierenden Berechnungen teilt, wird in 2 Funktionen gefasst:
    - AtomicActionBegin(ID)
    - AtomicActionEnd(ID)
  - Atomare Operationen werden auch als kritische Abschnitte bezeichnet.
  - Identifikator ID erlaubt die Unterscheidung verschiedener kritischer Abschnitte.

Kritischer Abschnitt



Triviale Synchronisation

- Als einfaches Verfahren eignet sich die Sperrung von Unterbrechungen.
  - AtomicActionBegin() { DisableInterrupts(); }
  - AtomicActionEnd() { EnableInterrupts(); }

- Bewertung
  - Rechnender Prozess kann den Prozessor monopolisieren.
  - Lange kritische Abschnitte können die Reaktion auf anstehende Unterbrechungen untolerierbar verzögern.
    - besonders für Echtzeitanwendungen untolerierbar
  - Der Mechanismus ist für Mehrprozessorsysteme ungeeignet.
  - Die Differenzierung zwischen zwei Abschnitten ist nicht möglich.

Prozess-Synchronisation durch Sperrung

- Verwendung der Maschinen-Instruktion LOCK.
  - Wartende Prozesse werden in eine aktive Warteschleife gezwungen (engl. busy waiting loop).
    - auch als Spin-Locks bezeichnet

- Bewertung
  - Aktiv wartende Prozesse verschwenden nutzlos Prozessor-Zyklen.
  - Bei Monoprozessor-Systemen können Prozesse bei unfairem Scheduling aushungern (engl. starvation).
    - Situation gleicht einem System-Stillstand.

Weitere Verfahren und Bewertung

- Weitere Ansätze zur Synchronisation durch Sperrung
  - Token-Passing Verfahren
    - Wettbewerb wird durch Weitergabe eines Token vermieden.
    - Nur ein Prozess darf das Token besitzen und weitergeben.
  - Dekker-Algorithmus
    - Erweiterung des Token Passing Verfahrens
    - Interessierte Prozesse reihen sich in einer Warteschlange auf.
    - Prozess gibt das Token an einen dedizierten Nachfolger weiter
  - Peterson Algorithmus
    - Variante des Dekker-Algorithmus, verwendet aber kein Token-Passing
- Bewertung
  - In allen Algorithmen ist derjenige Prozess Sieger, der zuerst sein Interesse bekundet hat.
  - Alle wartenden Prozesse befinden sich in einer aktiven Warteschleife.

Semaphore, Dijkstra 1968

- erster praxistauglicher Mechanismus zur Synchronisation, heute in fast allen Betriebssystemen anzutreffen
  - geeignet für kritische Abschnitte und
  - zur Kommunikation zwischen speichergekoppelten Prozessen

- Definition von Semaphoren
  - 2 Funktionen P(S) = passieren und V(S) = verlassen
    - vergleichbar mit AtomicActionBegin und AtomicActionEnd
  - S = Variable vom Typ Semaphor
    - für alle gemeinsam benutzten Daten eines kritischen Abschnittes muss eine Variable vom Typ Semaphor deklariert werden.

Vorteile von Semaphoren

- Wartende Prozesse werden blockiert, anstatt in einer aktiven Wartestellung zu verweilen.
  - P(S) und V(S) werden als atomare Operationen implementiert und in den Nukleus platziert.
  - P(S) und V(S) verwenden ggfs. die Instruktionen LOCK und UNLOCK für Unterbrechungen.

Semaphore Variante 1: Binäre Semaphore

- Binäre Semaphore sind Tupel (Z,W) mit

  Z = [true|false] und W = $\{P_1,P_2,\ldots,P_n\}$

  - Z entspricht dem Zustand der Blockade
    - Z=true: passieren erlaubt
  - W ist eine Liste von Prozessen, die potenziell in der P-Operation blockiert sind.

Binäre Semaphore: Zusicherungen für P(S) und V(S)

- Sei P_a der Prozess, der in einen Funktionsaufruf von P oder V involviert ist: $W = \{P_1,\ldots,P_a,\ldots,P_n\}$.

- Seien Z‘ und W‘ die Werte von Z und W vor der Ausführung der Operation.

Binäre Semaphore: Zusicherungen für P(S) und V(S)

$P(S):$

$(1): [(Z' = \text{true}) \wedge (W' = \varnothing)] \rightarrow P(S) \rightarrow [(Z = \text{false}) \wedge (W = \varnothing)]$

$(2): (Z' = \text{false}) \rightarrow P(S) \rightarrow [(Z = \text{false}) \wedge (W = \{P_a\} \cup W') \wedge (P_a = \text{blockiert})]$

$V(S):$

$(1): [(Z' = \text{true}) \wedge (W' = \varnothing)] \rightarrow V(S) \rightarrow [(Z = \text{true}) \wedge (W = \varnothing)]$

$(2): [(Z' = \text{false}) \wedge (W' = \varnothing)] \rightarrow V(S) \rightarrow [(Z = \text{true}) \wedge (W = \varnothing)]$

$(3): [(Z' = \text{false}) \wedge (W' \neq \varnothing)] \rightarrow V(S) \rightarrow [(Z = \text{false}) \wedge (W = W' \backslash \{P_x\}) \wedge (P_x = \text{bereit})], P_x \in W$

Beispiel für kritische Abschnitte

```
boolean S = true;  // Semaphor

process A {
    :
    P(S)
    :
    V(S)
    :
}
```

kritischer Abschnitt

```
process B {
    :
    P(S)
    :
    V(S)
    :
}
```

kritischer Abschnitt

Einfache Kommunikationsbeziehung: Erzeuger/Verbraucher

```
boolean S = false;  // Semaphor

process Erzeuger {
    :
    V(S)
    :
    :
    :
}
```

```
process Verbraucher {
    :
    :
    :
    P(S)
    :
}
```

Einfache Kommunikationsbeziehung: Erzeuger/Verbraucher

- Erzeuger signalisiert durch V(S) die Bereitstellung von Daten.
- Verbraucher wartet in der P(S) Operation auf die Bereitstellung.
  - Verbraucher konsumiert die Daten, nachdem das Ereignis (V(S)) eingetreten ist.
- Das Semaphor wird im Erzeuger/Verbraucher Fall mit false initialisiert.
- Bewertung
  - Wird ein Ereignis nicht schnell genug konsumiert, können Ereignisse durch zwei aufeinander folgende V(S) Operationen verloren gehen.
    - fatal, wenn Erzeuger z.B. ein Zeitgeber ist
    - z.B. Reader/Writer Problem

Semaphore Variante 2: Allgemeine Semaphore

- Allgemeine Semaphore sind Tupel (Z,W) mit

  $Z = [0,1,\ldots n]$ und $W = \{P_1,P_2,\ldots,P_n\}$

  - Z entspricht einem Zähler der Prozesse, die sind in einem kritischen Abschnitt befinden
    - Z wird üblicherweise mit 1 initialisiert, bei Erzeuger/Verbraucher mit 0.
  - W ist eine Liste von Prozessen, die potenziell in der P(S)-Operation blockiert sind.

Semaphore Variante 2: Allgemeine Semaphore

- Bedeutung
  - Bei jeder P(S) Operation wird der Zähler dekrementiert.
    - bei nicht positivem Z wird der ausführende Prozess blockiert
  - Bei jeder V(S) Operation wird der Zähler inkrementiert.
    - bei negativem Z wird der am Semaphor wartenden Prozess aktiviert

- Bewertung
  - In der Praxis findet man fast ausschließlich allgemeine Semaphore, da sie kaum Mehraufwand bedeuten.

Allgemeine Semaphore: Zusicherungen für P(S) und V(S)

$$P(S):$$

$$(1):[(Z'>0)\wedge(W'=\varnothing)]\rightarrow P(S)\rightarrow[(Z=Z'-1)\wedge(W=\varnothing)]$$

$$(2):(Z'\leq 0)\rightarrow P(S)\rightarrow[(Z=Z'-1)\wedge(W=\{P_a\}\cup W')\wedge(P_a=\text{blockiert})]$$

$$V(S):$$

$$(1):[(Z'\geq 0)\wedge(W'=\varnothing)]\rightarrow V(S)\rightarrow[(Z=Z'+1)\wedge(W=\varnothing)]$$

$$(2):[(Z'<0)\wedge(W'\neq\varnothing)]\rightarrow V(S)\rightarrow[(Z=Z'+1)\wedge(W=W'\backslash\{P_x\})\wedge(P_x=\text{bereit})],P_x\in W'$$

- Die übrigen Fälle können nicht eintreten.
- Aber: Semaphor-Zähler kann überlaufen!

Implementierung in Pseudocode

```
class Semaphor {
    int Z;
    List[] W; // PCB, process control
    block
}

void P(Semaphor S) {

    PCB Pa;  // aktiver Prozess

    if (S.Z<=0) {  // blockieren
            S.W.add(Pa);
            Dispatcher.block();
            Dispatcher.assign();
    }
    S.Z -= 1;  // dekrement
}
```

```
void V(Semaphor S) {

    PCB Px;  // wartender Prozess

    if (S.Z<0) {  // bewerben
            Px=S.W.getFirst();
            Dispatcher.ready(Px);
            Dispatcher.resign();
            Dispatcher.assign();
    }
    S.Z += 1;  // inkrement
}
```

optional: Prozesswechsel auslösen

Reader-Writer Problem

- Reader und Writer bearbeiten einen gemeinsamen Datenbereich.
  - Beliebig viele Reader und Writer greifen konkurrierend auf Daten zu.
    - Reader lesen ausschließlich
    - Writer schreiben ausschließlich
    - Writer beanspruchen exklusiven Zugriff

Übung: Reader-Writer Problem

- Implementieren Sie das Reader-Writer Problem auf der Basis von allgemeinen Semaphoren.
  - Es sollen eine beliebige Zahl von Readern und Writern teilnehmen.
  - Implementieren Sie beide als Threads, die eine zufällige Zeit warten und dann einen lesenden bzw. schreibenden Zugriff auf den Speicher durchführen.
  - Jeder Reader bzw. Writer soll einen zufällig gewählten Datenbereich adressieren.
- Weisen Sie die Funktionsfähigkeit durch geeignete Messreihen nach.
- Finden Sie eine optimale Strategie.

Hinweise zur Übung

- Legen Sie den Datenbereich als Array von Byte an.
- Verwenden Sie mehrere Semaphore.
- Sie finden das reader-Writer Problem auch in der Literatur.

Synchronisation über Monitore

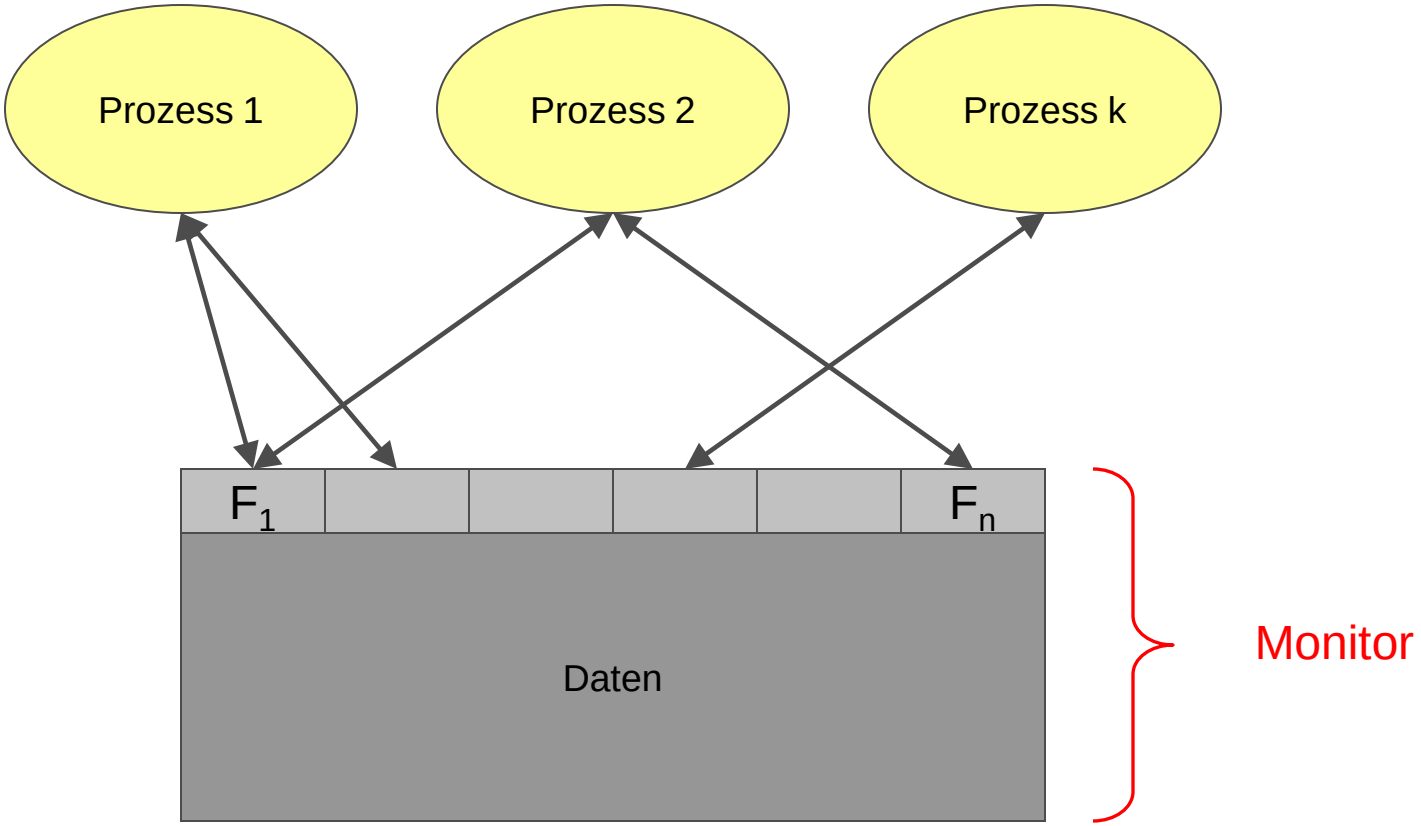
- Warum Monitore? Semaphore weisen eine Reihe von Schwächen auf:
  - Daten und dazugehöriger Code sind bei Semaphor-Mechanismen durch den Programmcode verstreut.
    - Programm wird unverständlich.
  - Kritische Abschnitte sind oft bedingt und damit wenig überschaubar.
  - Implementierungen sind fehleranfällig und instabil.

Was sind Monitore?

- Ein Monitor ist eine Datenstruktur mit darauf definierten Operationen, die sich zeitlich gegenseitig ausschließen (Hoare, 1974).

- Idee eines Monitors
  - jedem gemeinsam von mehreren Prozessen genutzte Datenstruktur mit den darauf definierten Zugriffsfunktionen $F_1,\ldots,F_n$ zusammenfassen – dem Monitor
  - logische Unteilbarkeit der Funktionen $F_1,\ldots,F_n$ sicherstellen
  - Zugriffe zu den Daten nur durch die Funktionen $F_1,\ldots,F_n$ des Monitors zulassen

Was sind Monitore?



Vorteile von Monitoren gegenüber Semaphoren

- Die Datenstruktur wird mit allen Zugriffsfunktionen zusammenhängend beschrieben.
  - Es wird ausgeschlossen, dass neben dem Monitor noch weitere Funktionen existieren, die auf die Daten zugreifen.

- Monitore kapseln einen Entwurf. Die interne Verarbeitung der Daten ist transparent.
  - Erst die Änderungen der Funktionsschnittstelle (F_1,…,F_n) führen zu Änderungen der aufrufenden Programme.

Monitore: Eintritt in kritische Abschnitte

- Für den Eintritt in einen kritischen Abschnitt definieren Monitore eine sog. Condition-Variable (CV).
  - Condition-Variable sind nur innerhalb des Monitors definiert.
- Eine Condition-Variable speichert Hinweise auf alle Prozesse, die auf die Erfüllung der zugeordneten Bedingung warten.
- 3 Operationen auf Condition-Variablen
  - CV.wait() bzw, CV.wait(Priorität)
  - CV.signal() bzw. CV.signal(Priorität)
  - CV.status()

Monitore: Operationen auf Condition-Variablen

- wait(), wait(Priorität)
  - wait blockiert einen Prozess, bis die zugeordnete Bedingung (CV) erfüllt ist. Der Prozess wird in die Warteschlange der CV aufgenommen.
  - Ist eine Priorität angegeben, wird der Prozess an eine geeignete Stelle in der Warteschlange eingefügt.

- signal(), signal(Priorität)
  - Wenigstens ein Prozess, der sich zum Zeitpunkt des Aufrufs in der Warteschlange der CV befindet, wird ausgekettet und aktiviert.
  - Ist eine Priorität angegeben, wird ein Prozess mit der passenden Priorität gewählt.

- status()
  - Mit status wird der momentane Zustand einer CV erfragt.
  - status liefert LEER oder NICHT_LEER

Monitore: Eintritt in den Monitor

- notwendige Informationen bei konkurrierenden Zugriffen
  - Zustand des Monitors: FREI oder BELEGT
  - alle Prozesse kennen, die bei belegtem Monitor auf einen Zutritt warten

- Datenstruktur für den Eintritt
  - für jeden Monitor wird ein Tupel M=(Z,W) gespeichert
    - Z=Zustand: FREI oder BELEGT
    - W=Menge der Prozesse, die auf den Zutritt zum Monitor warten
    - Initialwert: M=(FREI,Ø)

Spezifikation von Monitoreintritt und Monitoraustritt

- Sei P_a der Prozess, der eine Operation ausführt.
- Seien Z‘ und W‘ die Werte von Z und W vor der Ausführung des Eintritts bzw. Austritts in den Monitor.
- Die Spezifikation von Eintritt und Austritt ist äquivalent zu den Operationen P(S) und V(S) für Semaphore.

Spezifikation von Monitoreintritt und Monitoraustritt

Monitoreintritt

(1)

$$[(Z' = \text{FREI}) \wedge (W' = \varnothing) \rightarrow \text{EINTRITT} \rightarrow [(Z = \text{BELEGT}) \wedge (W = \varnothing)]$$

(2)

$$[(Z' = \text{BELEGT}) \rightarrow \text{EINTRITT}$$
$$\rightarrow [(Z = \text{BELEGT}) \wedge (W = \{P_a\} \cup W') \wedge (P_a = \text{BLOCKIERT}]$$

Monitoraustritt

(1)

$$[(Z' = \text{BELEGT}) \wedge (W' = \varnothing) \rightarrow \text{AUSTRITT} \rightarrow [(Z = \text{FREI}) \wedge (W = \varnothing)]$$

(2)

$$[(Z' = \text{BELEGT}) \wedge (W' \neq \varnothing) \rightarrow \text{AUSTRITT}$$
$$\rightarrow [(Z = \text{BELEGT}) \wedge (W = W' \backslash \{P_x\}) \wedge (P_x = \text{BEREIT})], P_x \in W'$$

Spezifikation der CV Operationen

- Sei CV‘ der Wert der Condition-Variable vor der Ausführung einer CV-Operation.
- Sei CV.W={P_1,…,P_k} die Menge von Prozessen, die auf den Eintritt einer mit CV assoziierten Bedingung CV.B warten.
- Kann ein Prozess P_a den Monitor über die wait-Operation nicht betreten, wird er blockiert.

Spezifikation der CV Operationen: status()

CV.status()

(1)

$$(CV.W' = \varnothing) \rightarrow CV.status()$$

$$\rightarrow [(CV.W' = \varnothing) \wedge (STATUS = LEER)]$$

(2)

$$(CV.W' \neq \varnothing) \rightarrow CV.status()$$

$$\rightarrow [(CV.W' \neq \varnothing) \wedge (STATUS = NICHT\_LEER)]$$

Spezifikation der CV Operationen: P_a ruft wait() auf

CV.wait()

(1)

$$[\neg CV.B \wedge (W' = \varnothing) \wedge (Z' = BELEGT)] \rightarrow CV.wait()$$
$$\rightarrow [\neg CV.B \wedge (CV.W = \{P_a\} \cup CV.W') \wedge$$
$$(P_a = BLOCKIERT) \wedge (W = \varnothing) \wedge (Z = FREI)]$$

P_a gibt Monitor frei, wenn kein Prozess in W‘ wartet.

(2)

$$[\neg CV.B \wedge (W' \neq \varnothing) \wedge (Z' = BELEGT)] \rightarrow CV.wait()$$
$$\rightarrow [\neg CV.B \wedge (CV.W = \{P_a\} \cup CV.W') \wedge (W = W'\backslash\{P_x\}, P_x \in W') \wedge$$
$$(P_x = BEREIT) \wedge (Z = BELEGT)]$$

Ein wartender Prozess wird aktiviert.

Spezifikation der CV Operationen: P_a ruft signal() auf

CV.signal()

Variante 1:

kein Besitzwechsel, Reaktivierung höchstens eines wartenden Prozesses

(1)

$$[CV.B \wedge (CV.W' = \varnothing)] \rightarrow CV.signal() \rightarrow [CV.B \wedge (CV.W = \varnothing)]$$

(2)

$$[CV.B \wedge (CV.W' \neq \varnothing)] \rightarrow CV.signal()$$
$$\rightarrow [CV.B \wedge (CV.W = CV.W' \backslash \{P_x\}, P_x \in CV.W') \wedge (W = W' \cup \{P_x\})]$$

Spezifikation der CV Operationen: P_a ruft signal() auf

CV.signal()

Variante 2:

kein Besitzwechsel, Reaktivierung aller wartenden Prozesse

(1)

$$[CV.B \wedge (CV.W' = \varnothing)] \rightarrow CV.signal() \rightarrow [CV.B \wedge (CV.W = \varnothing)]$$

(2)

$$[CV.B \wedge (CV.W' \neq \varnothing)] \rightarrow CV.signal()$$

$$\rightarrow [CV.B \wedge (CV.W = \varnothing) \wedge (W = W' \cup CV.W')]$$

Spezifikation der CV Operationen: P_a ruft signal() auf

CV.signal()

Variante 3:

Besitzwechsel vom signalisierenden auf den signalisierten Prozess

(1)

$$[CV.B \wedge (CV.W' = \varnothing) \rightarrow CV.signal() \rightarrow [CV.B \wedge (CV.W = \varnothing)]$$

(2)

$$[CV.B \wedge (CV.W' \neq \varnothing) \rightarrow CV.signal()$$
$$\rightarrow [CV.B \wedge (CV.W = CV.W' \backslash \{P_x\}, P_x \in CV.W') \wedge (P_x = BEREIT)$$
$$\wedge (W = W' \cup \{P_a\}) \wedge (P_a = BLOCKIERT)]$$

Übung

- Bewerten Sie die drei Varianten der signal()-Operation. Welche erscheint am sinnvollsten?
- Begründen Sie ihre Entscheidung.

Übung

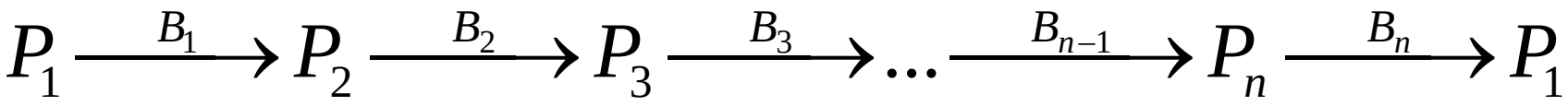
- Implementieren Sie das Monitor-Konzept ohne die Verwendung von Prioritäten.
  - Setzen Sie das Konzept in Java um.
  - Hinweis: der gegenseitige Ausschluss von Methodenaufrufen lässt sich mit Semaphoren realisieren.
- Entwickeln Sie für Ihre Implementierung einen Demonstrator und optimieren Sie den Durchsatz:
  - Implementieren Sie ein System mit 4 parallel verfügbaren, gleichen Druckern und ein einer beliebigen Menge von Prozessen (0-20), die unterschiedliche Datenmengen drucken wollen.
    - Datenmengen können zwischen 1 und 10 Einheiten groß sein.
    - Das Drucken einer Einheit auf einem Drucker dauert genau 1 Sekunde.
    - Erzeugen Sie spätestens nach 2 Sekunden einen neuen Prozess mit einer zufällig gewählten Datenmenge.

Synchronisationsfehler

- Synchronisationsfehler können als Folge inkorrekt angewendeter Synchronisation auftreten. Ergebnis ist eine Ablaufinkonsistenz oder eine Verklemmung.

- Eine Verklemmung (engl. deadlock) ist ein Zustand, in dem alle beteiligten Prozesse wechselseitig auf den Eintritt von Bedingungen warten, die nur durch Prozesse dieser Gruppe selbst hergestellt werden können.
  - Prozesse bleiben dauerhaft blockiert, wenn die Verklemmung nicht durch Eingreifen von außen behoben wird.
  - Man spricht von einer totalen Verklemmung, wenn alle Prozesse des Systems betroffen sind.

Formale Beschreibung: Zirkulare Wartebedingung



- charakteristisch für Synchronisationsfehler: Zeitabhängigkeit
  - Fehler sind nur schwer reproduzierbar.
  - Fehler können nicht durch systematisches Testen gefunden werden.

- Wie kann man Verklemmungen vermeiden?

Beispiel: fehlerhafte kritische Abschnitte

```
Semaphor S =
    true;
A() {
    :
    :
    while (…) {
        P(S)
        f1;
        V(S);
    }
}
```

```
B() {
    :
    :
    while (…) {
        P(S)
        f2;
    }
}
```

```
C() {
    :
    :
    while (…) {
        f3;
        V(S);
    }
}
```

Beispiel: fehlerhafte kritische Abschnitte

- Die Prozesse A, B und C bearbeiten zyklisch einen gemeinsamen Datenbereich durch die Zugriffsfunktionen f1, f2 und f3, die innerhalb kritischer Abschnitte liegen.
- A sperrt den kritischen Abschnitt korrekt.
- Bei B fehlt die V(S) Operation und bei C die P(S) Operation.
  - C tritt unkoordiniert in den kritischen Abschnitt ein.
  - B gibt den Zugriff zu den gemeinsamen Daten am Ende des kritischen Abschnitts nicht frei.

- Ergebnis: inkonsistente Daten, da C parallel zu A oder B auf die gemeinsamen Daten zugreifen kann.

Auflösen von Verklemmungen (Holt, 1972)

- Zustandsübergänge und Ressourcen-Anforderungen eines Systems von Prozessen können als Kanten, Zustände als Knoten eines gerichtetem, gewichteten Graphen formuliert werden.
  - Erkennung von Verklemmungen lässt sich auf die Betrachtung von Pfaden in diesem Graphen reduzieren.
- Zustände werden klassifiziert:
  - blockiert
    - Ein Zustand ist blockiert, wenn es keinen Übergang zu einem Folgezustand gibt.
  - verklemmt
    - Ein Zustand ist verklemmt, wenn jeder Zustandsübergang zu einem blockierten Zustand führt.
  - sicher
    - Ein Zustand ist sicher, wenn kein Zustandsübergang in einen verklemmten Zustand führt.

Selbstkontrolle

1. Warum können kritische Abschnitte in der Praxis nicht durch die Prozessoroperationen LOCK/UNLOCK realisiert werden?
2. Was sind Semaphore?
3. Wie können mehrere Ressourcen mit Semaphor-Variablen verwaltet werden?
4. Erläutern Sie das Erzeuger-Verbraucher System mit allgemeinen Semaphoren.
5. Was ist ein Monitor?
6. Warum sind Implementierungen mit Monitoren üblicherweise stabiler als mit Semaphoren?
7. Warum wird der Monitor freigegeben, wenn in der Warteschlange kein Prozess aber in der Condition-Variablen ein Prozess wartet?

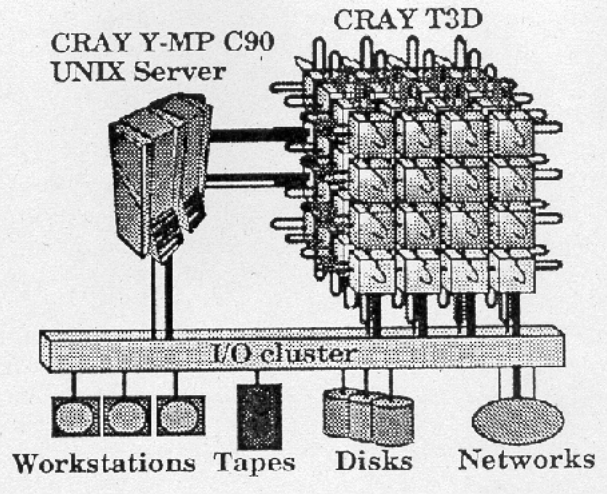
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Betriebssysteme
Speicherverwaltung



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Überblick

- Speicherorganisation in Segmenten
  - statisch
  - dynamisch
- Adressräume
  - Swapping
  - virtuelle Adressierung
  - Seitenverdrängungsverfahren

Speicherbedarf für Prozesse

- Untere Schichten des Betriebssystems werden an einen festen Ort geladen.
  - Speicher kann direkt zugewiesen werden.
  - z.B. Nukleus
- Höhere Schichten und Anwendungen…
  - haben eine beschränkte Lebensdauer
  - benötigen dynamische Speicherzuweisung

Dynamische Speicherzuweisung

- Dynamische Speicherzuweisung erfordert die Verwaltung des Arbeitsspeichers.
  - implementiert in Betriebssystemfunktionen:
    - allocate
    - free
- Speicherverwaltung wird unterschieden:
  - starre Segmentierung
  - dynamische Segmentierung

Dynamische Speicherzuweisung

- starre Segmentierung
  - verfügbarer Speicher wird in starre Segmente einer vorgegebenen Größe unterteilt
  - vorteilhaft, wenn überwiegend gleiche Speichergrößen benötigt werden
    - Segmentgröße entspricht der größten Speicheranforderung.

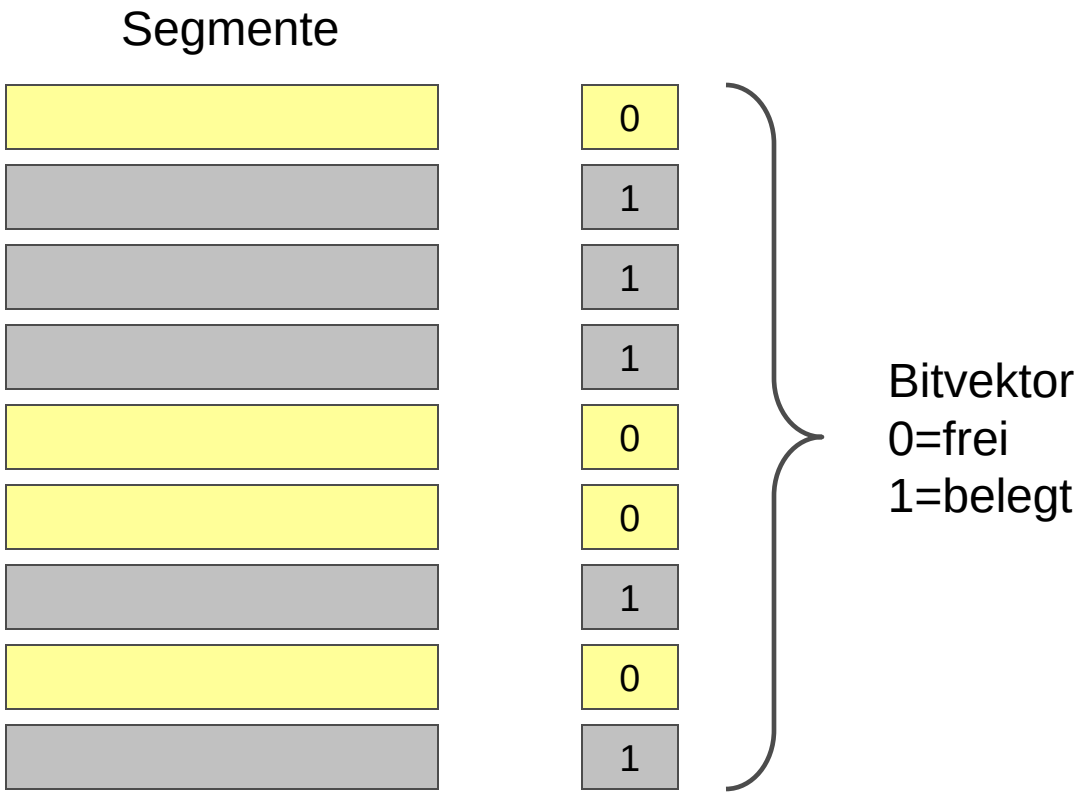
- dynamische Segmentierung
  - Speicher wird stückweise in der Reihenfolge der Anforderungen zugeteilt.
  - dynamisch: Speichersegmente entsprechen genau der Anforderung
  - vorteilhaft bei stark unterschiedlichen Speicheranforderungen

Starre Segmentierung

- Speicherverwaltung einfach zu implementieren
  - int address = allocate()
  - void free(int address)

- Organisation der Segmente erfolgt über einen Vektor von Bits
  - jedes Bit kennzeichnet den frei/belegt Zustand eines Segments
  - Implementierung erfolgt über einen Monitor
    - stellt allocate, free und Zustandsinformationen bereit

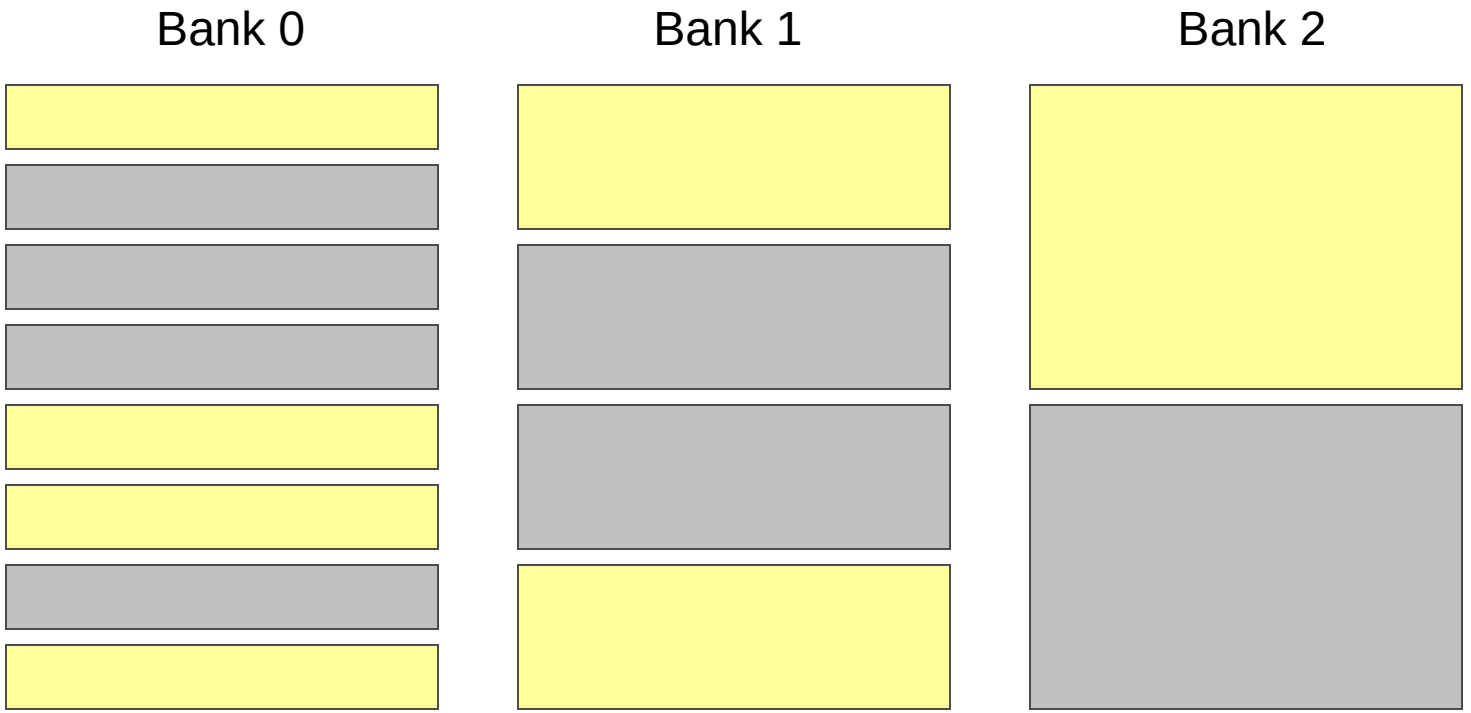
Speicherverwaltung bei starrer Segmentierung



Optimierung: mehrere feste Größen

- Speicher wird in Bänken organisiert. Jede Bank speichert Segmente einer festen Größe.
- Speicheranforderungen werden auf geeignete Segmente abgebildet.
- Implementierung:
  - int address = allocate(int size)
  - void free(int address)

Optimierung: mehrere feste Größen



Starre Segmentierung: Bewertung

- Speicher wird auch beim Einsatz von Bänken schlecht genutzt.
- Ein Zusammenfassen unbenutzter Segmentteile ist nicht möglich.
- Bemerkung:
  - Man spricht hier von der Fragmentierung des Speichers.

Dynamische Segmentierung

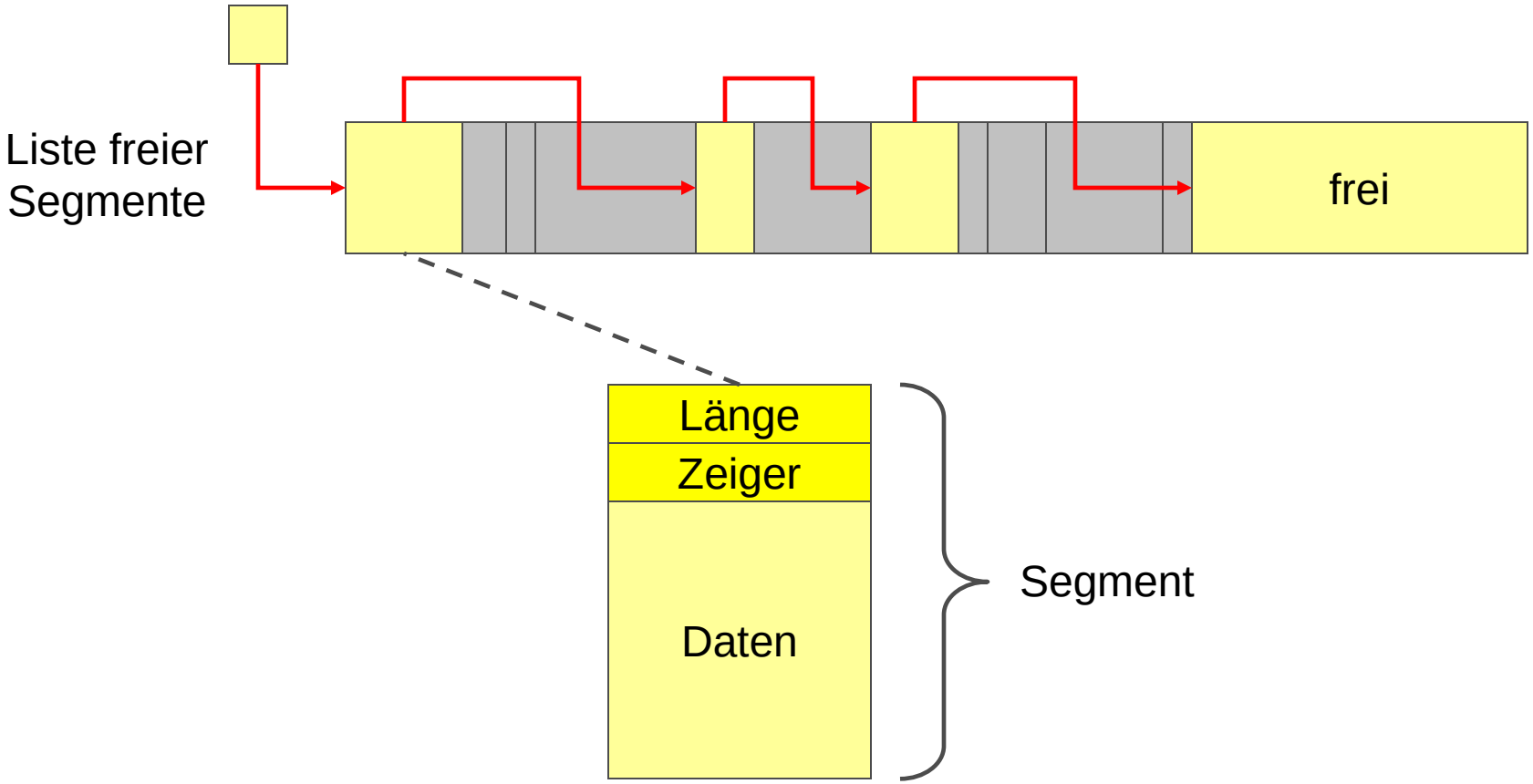
- Arbeitsweise
  - zu Beginn liegt ein freies Segment im Speicher: Segment entspricht der Speichergröße
  - bei einer Anforderung wird ein freies Speichersegment unterteilt:
    - ein neues belegtes Segment
    - ein verbleibendes freies Segment
  - freie Segmente werden in einer Liste verwaltet
    - nebeneinander liegende freie Segmente werden zusammengefasst
    - die Mindestgröße eines Segments wird begrenzt

Dynamische Segmentierung

- Implementierung
  - int address = allocate(int size)
  - void free(int address, int size)

- Verwaltung eines freien Speichersegments
  - jedes freie Segment enthält einen Zeiger auf das nächste im Speicher liegende freie Segment

Organisation bei dynamischer Segmentierung



Verfahren für die Zuweisung von freien Segmenten

- First Fit
  - die Liste freier Segmente ist unsortiert
  - bei einer Anforderung wird das erste passende Segment der Liste gewählt
- Best Fit
  - die Liste freier Segmente ist unsortiert oder nach der Segmentgröße aufsteigend sortiert
  - bei einer Anforderung wird das kleinste passende Segment der Liste gewählt

Übung: dynamische Segmentierung (1)

- Implementieren Sie dynamische Speichersegmentierung
  - Realisieren Sie First Fit und Best Fit.
- Vergleichen Sie die Speichereffizient beider Verfahren.
  - Implementieren Sie dafür eine Testklasse, die Speicher in zufälliger Größe anfordert.
  - Messen Sie mit den angegebenen Metriken.
  - Ihre Testklasse soll dann abbrechen, wenn eine Anforderung nicht mehr erfüllt werden kann. Der Speicher ist damit voll.

Übung: dynamische Segmentierung (2)

- Randbedingungen:
  - freie Segmente werden in einer linearen Liste organisiert
  - nach 2 Anforderungen werden zufällig 1-2 gewählte Segmente freigegeben
    - nebeneinander liegende freie Segmente werden zusammengefasst
  - minimale Segmentgröße: 5
  - Größe der Speicheranforderung: 1-100
  - Größen des Speichers: 10000, 1000000, 100000000
  - die Segmentliste ist für Best Fit aufsteigend sortiert

Übung: dynamische Segmentierung (3)

- Metriken
  1. benötigte Zeit für die Zuweisung eines Segments an eine Anforderung
  2. Anzahl der erfüllten Anforderungen
  3. Belegung des Speichers in Prozent
  4. Anzahl der freien Segmente
  5. größtes freies Segment

Bewertung: dynamische Segmentierung

- Best Fit liefert gegenüber First Fit kaum Vorteile.
  - Best Fit hat zusätzlich erheblich größeren Verwaltungsaufwand.
- Dynamische Segmentierung führt zu einer Zerstückelung des Speichers in sehr kleine, unzusammenhängende Segmente.
  - sog. externe Fragmentierung
- Fazit
  - bessere Verfahren haben weniger Verwaltungsaufwand und erzeugen keine Fragmentierung
  - Verbesserung: Buddy-Verfahren

verbesserte Segmentierung: Buddy-Verfahren

- Idee
  - Vorteile aus starrer und dynamischer Segmentierung vereinigen
  - Speicher wird als Segment der Größe 2^m verwaltet.
  - eine Anforderungen der Größe L erhält ein Speichersegment der Größe $2^{k-1} < L < 2^k$

- Segmentierung in 2er-Potenzen
  - benachbarte freie Segmente können leicht zusammengefasst werden
  - Segmentgröße lässt sich leicht auf eine Speicheradresse abbilden
  - Segmente der Größe 2^k können nur die Startadresse eines Vielfachen von 2^k haben
    - Voraussetzung: Adressierung beginnt bei 0

Buddy-Verfahren: Adressen berechnen

- Speichern eines Segments 2^k: Adresse besitzt beginnend mit dem LSB (least significiant bit) k Nullen
  - Wortbreite m entspricht der Speichergröße 2^m

$$\underbrace{*\ldots*}_{m-k}\underbrace{0\ldots0}_{k}$$

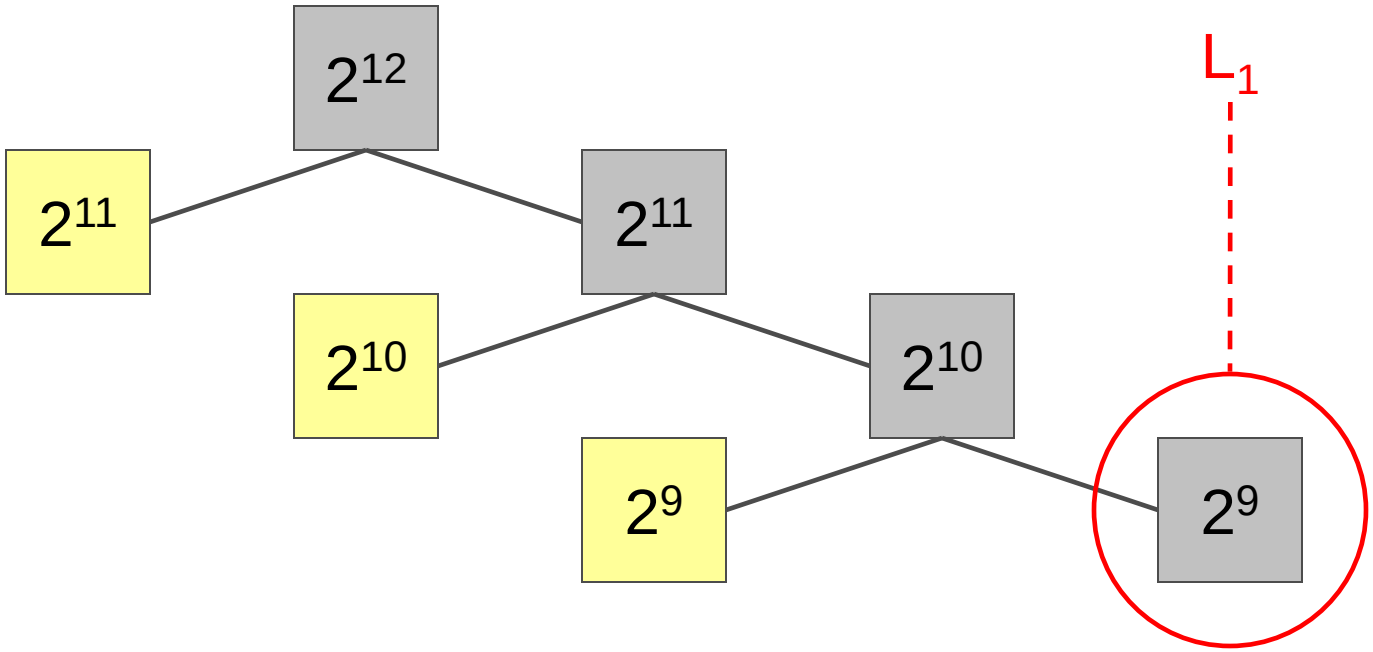
Buddy-Verfahren: freie Segmente verschmelzen

- freies Segment der Größe 2^k
  - Suchen in der Liste der freien Segmente nach einem Segment, dass sich nur an der Stelle k+1 unterscheidet
  - beide Segmente werden zu einem freien Segment der Größe 2^{k+1} vereinigt

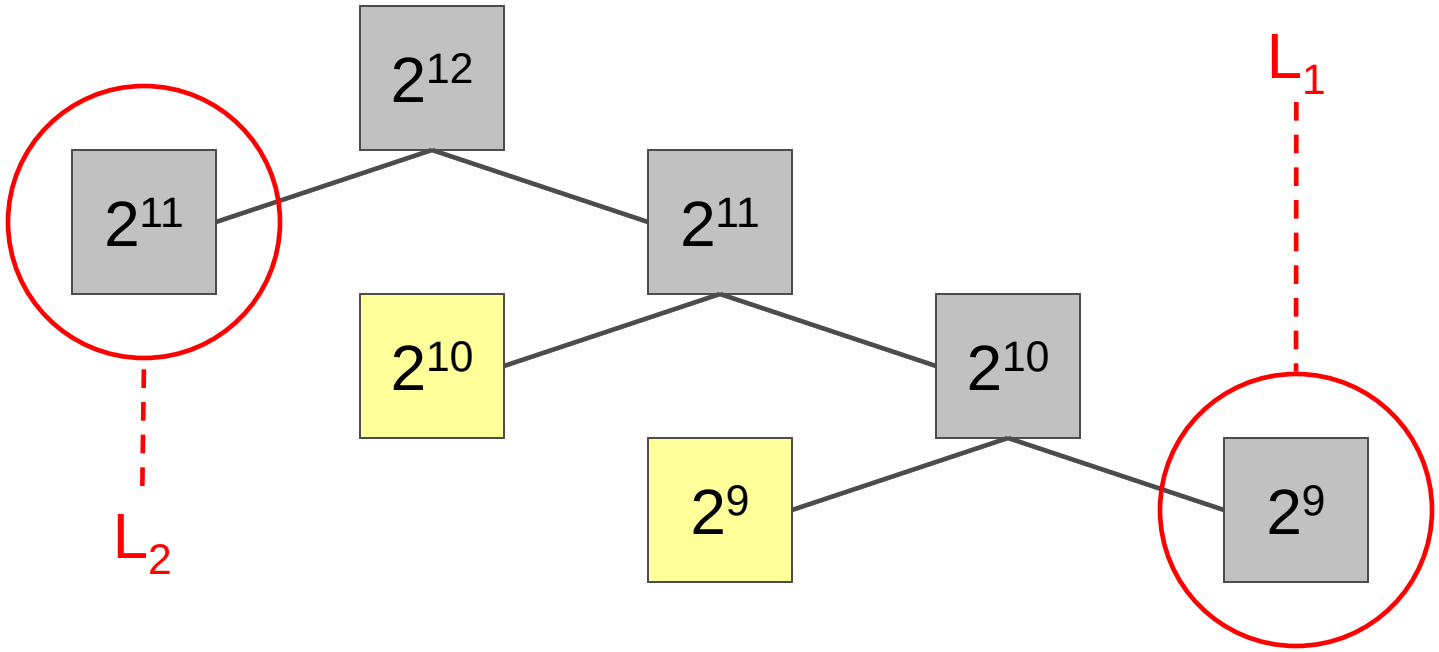
$$\underbrace{x\ldots x}_{m-k-1}\underbrace{0}_{}\underbrace{0\ldots 0}_{k} \oplus \underbrace{x\ldots x}_{m-k-1}\underbrace{1}_{}\underbrace{0\ldots 0}_{k} \rightarrow \underbrace{x\ldots x}_{m-k-1}\underbrace{0\ldots 0}_{k+1}$$

Beispiel: Speicher der Größe 2^{12}

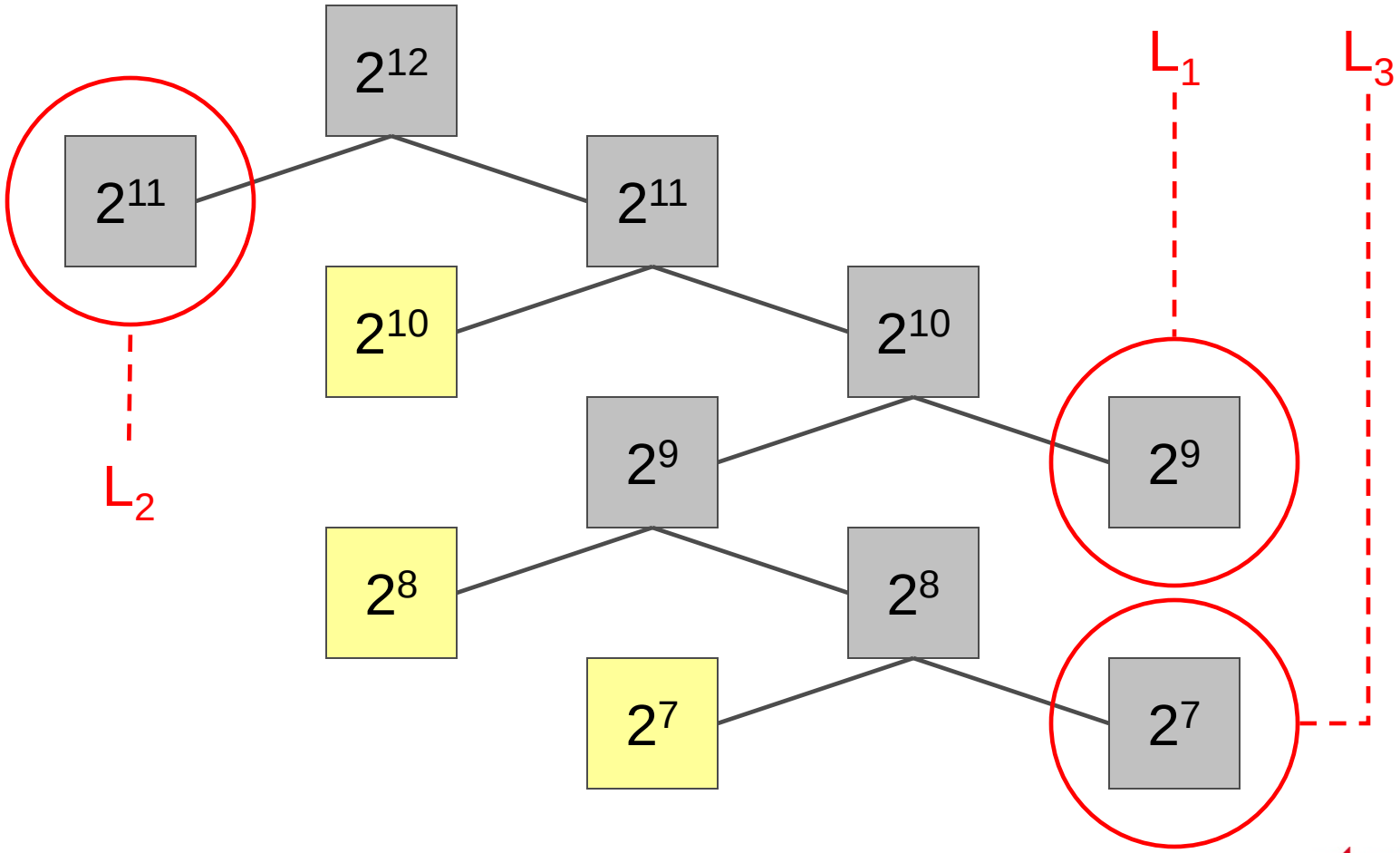
Beispiel: Anforderung L_1=450



Beispiel: Anforderung L_1=450, L_2=1500



Beispiel: Anforderung L_1=450, L_2=1500, L_3=78



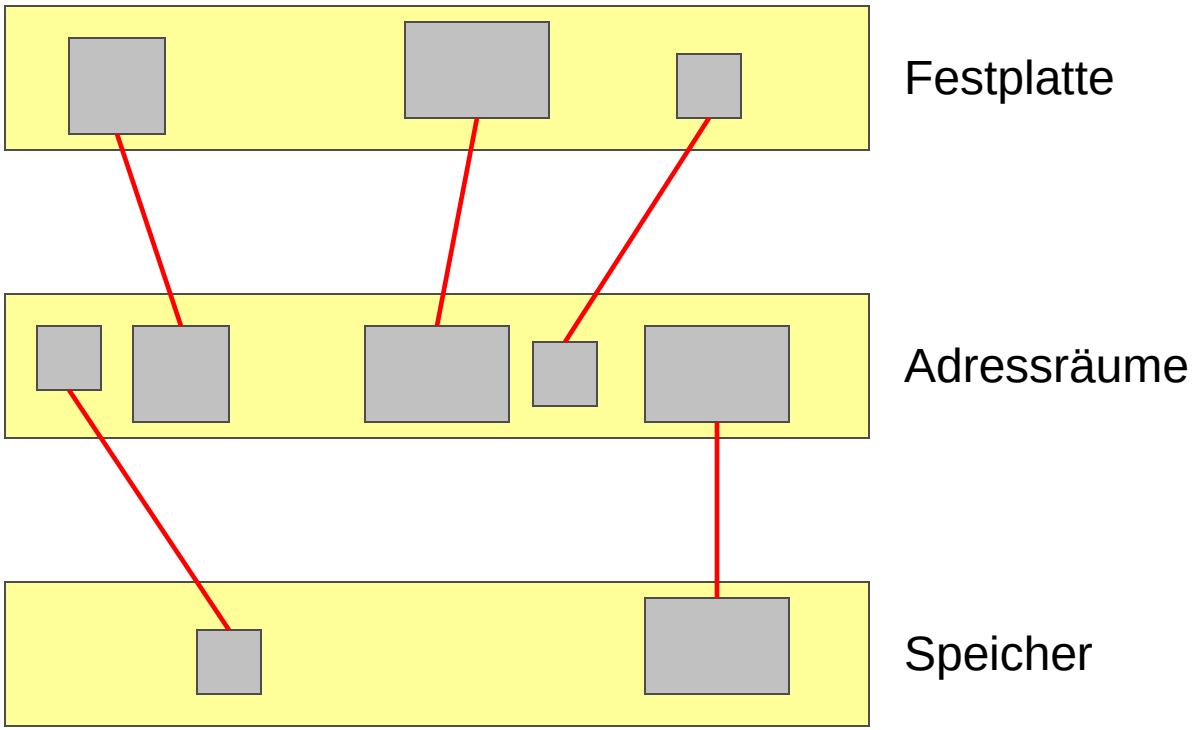
Adressräume

- Der Prozessor wird über Prozesse virtualisiert.
  - Konzept der Pseudoprozessoren, später Threads
- Idee der Adressräume
  - Virtualisierung des Speichers
  - effiziente Speicherverwaltung
- Virtualisierte Speicher werden als Adressräume bezeichnet.
- Ein Prozess wird als Tripel (S,C,P) aufgefasst.
  - S: Adressraum
  - C: sequentielles Programm im Adressraum
  - P: für dir Programmausführung zuständiger Thread
    - P enthält insbesondere den Programmzähler PC

Eigenschaften eines Adressraums

- Für Adressräume existieren keine festen Abbildungen auf physische Speicher.
- Zu einem Zeitpunkt können Teile des Adressraums im Speicher und andere auf der Festplatte liegen.
- Für große Bereiche eines Adressraums kann es zu einem Zeitpunkt überhaupt keine Entsprechung auf einem physischen Speichermedium geben.
- Adressräume benötigen die Unterstützung des Kerns.
  - Prozessorumschaltung wird im Nukleus um Speicherumschaltung erweitert.
  - Die Umschaltung wird als Multiplexing bezeichnet.

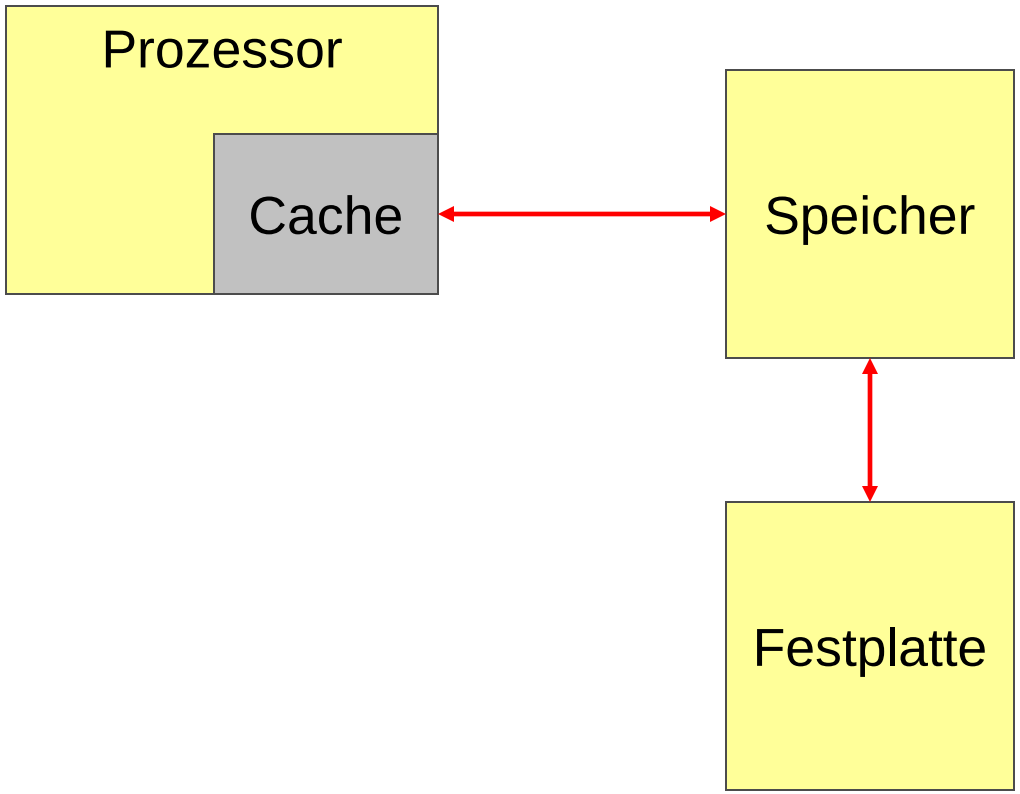
Abbildung eines Adressraums auf verschiedene Speichermedien



Organisation mehrerer Adressräume

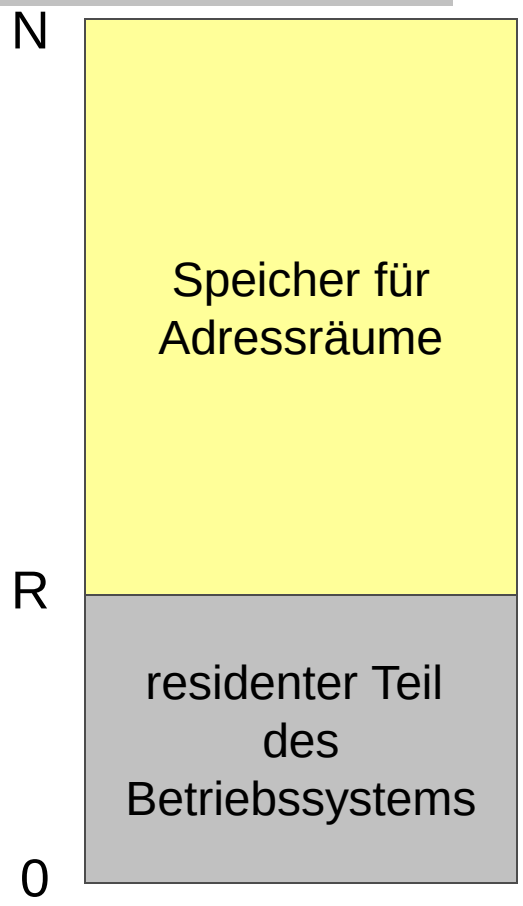
- Mehrere Adressräume können sich konkurrierend um die Zuweisung von Speichersegmenten bewerben.
- Speichersegmente müssen bei Bedarf von der Festplatte in den Speicher nachgeladen und nicht unmittelbar benötigte Seiten verdrängt werden.
- Prozessor, Cache, Speicher und Festplatte bilden die Speicherhierarchie eines Rechners.

Organisation mehrerer Adressräume



Swapping

- Speicherorganisation mit Adressräumen
  - Sei N die Größe des Speichers.
  - Sei R die Größe des residenten Teils des Betriebssystems.
  - Der Bereich [0..R[des Speichers bleibt für das Betriebssystem reserviert.
  - Der Bereich [R..N] steht für Adressräume zur Verfügung.



Erzeugung eines nicht-residenten Prozesses

- An die Erzeugung eines Prozesses wird nun die Erzeugung eines Adressraums geknüpft:

  int ProzessID = erzeugeProzess(int anfang, int laenge, int kellerLaenge, Info schedulerInfo)

  - Programmcode der Länge laenge wird aus dem Adressraum des Vaterprozesses von der Adresse anfang geladen.
  - Im Adressraum wird der Laufzeitkeller (engl. runtime stack) der Länge kellerLaenge erzeugt und der PCB (engl. process control block) initialisiert.
  - Über den Parameter schedulerInfo können Informationen an den Scheduler übermittelt werden.
    - Datentyp ist scheduler-spezifisch
    - z.B. Priorität

Laufzeitbereiche

- Voraussetzung für die Bearbeitung eines Prozesses ist, dass der Adressraum auf ein zusammenhängendes Segment des Arbeitsspeichers abgebildet werden kann.

- Definition: Laufzeitbereich, auch Laufbereich
  - Ein zusammenhängendes Arbeitsspeichersegment, dass einen konkreten Adressraum beherbergt, wird als Laufzeitbereich bezeichnet.

- Annahme:
  - Summe der Adressraum-Größen übersteigt den tatsächlich vorhandenen Arbeitsspeicher.
    - Wechsel von Adressräumen erforderlich (engl. swapping)

Aufgaben des Swapping Mechanismus

- Bestimmen von Anzahl und Größe benötigter Laufzeitbereiche
- Multiplexen der vorhandenen Laufbereiche unter den existierenden Adressräumen

- Optimierungskriterium für Speicher-Multiplexing
  - gute Prozessorauslastung bei minimalem Speicherbedarf

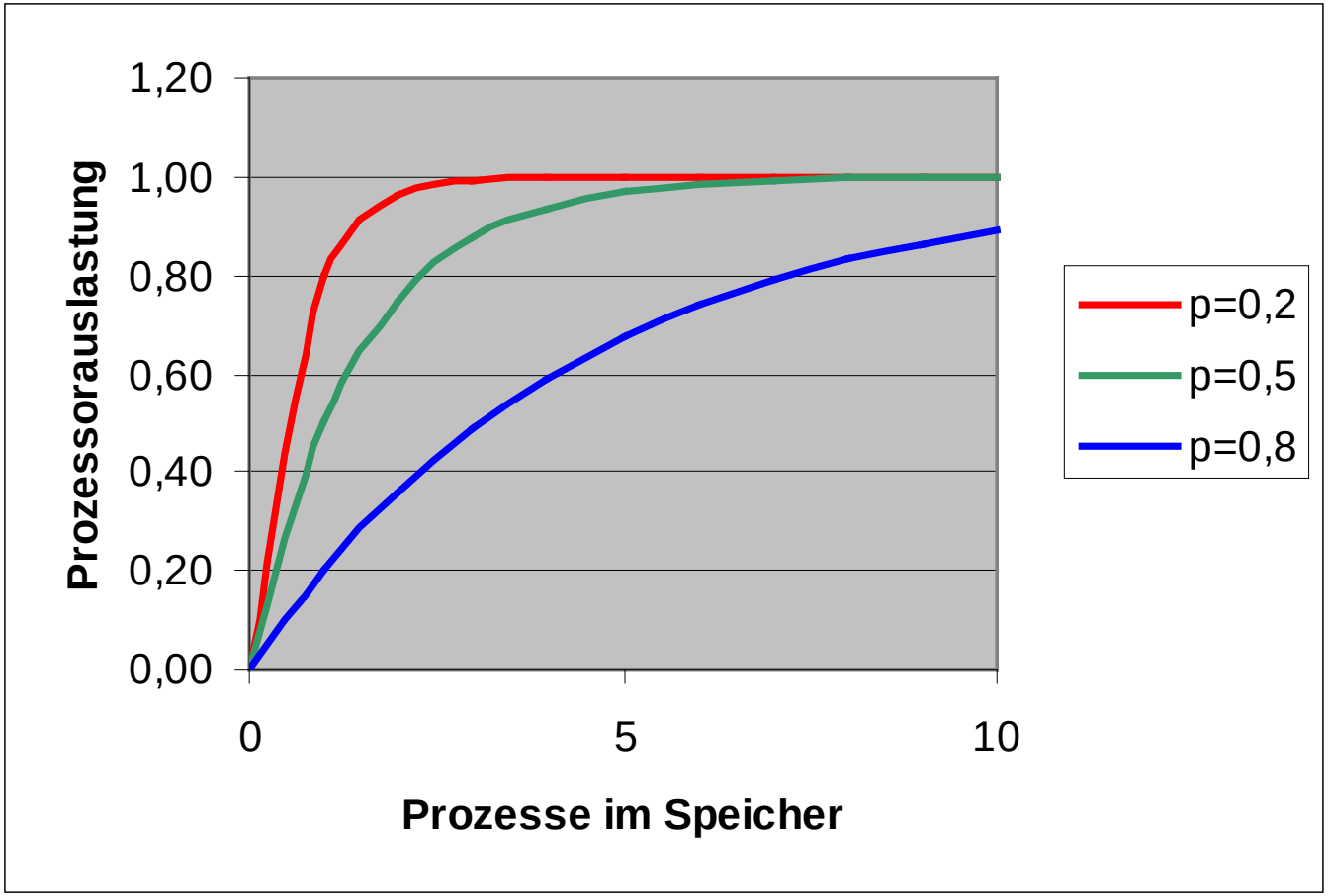
- Speicher-Multiplexer implementieren Medium-Term Scheduling.

Prozessorauslastung

- Sei n die Zahl der Prozesse.
- Sei p der Zeitanteil, den ein Prozess in einer blockierenden Operation verbringt.
  - p liegt im Intervall [0,1]
  - p wird auch als Wahrscheinlichkeit für eine Blockade angesehen.
  - Die Wahrscheinlichkeit, dass zu einem Zeitpunkt n Prozesse in einer Blockade sind ist p^n.

- Prozessorauslastung: $L_p(n) = 1 - p^n$

Prozessorauslastung bei E/A-Anteil p



Bewertung für p=0,5

- Bei einer gewünschten Mindestauslastung von 80% müssen wenigstens 3 Prozesse permanent im Speicher sein.
- Bei einem durchschnittlichen Speicherbedarf s wird kaum mehr als 3s Speicher benötigt.
- Rechtzeitiges Verdrängen macht einen größeren Speicher unnötig.

Swapping in einem festen Laufbereich

- Idee: gesamter freier Speicher wird einem Laufbereich zugeordnet.
  - Prozesse $P_1,\ldots,P_n$ werden zyklisch für die Dauer des processor bursts geladen und bei Prozesswechsel wieder auf die Festplatte verdrängt.

- Bewertung
  - sehr einfaches Verfahren
  - Prozessorauslastung ist leicht abschätzbar

- Berechnung der Prozessorauslastung:
  - b = mittlere Zeit eines processor bursts
  - w = mittlere Zeit einer E/A Operation
  - s = mittlere Größe eines Adressraums
  - r = Übertragungsrate zwischen Speicher und Festplatte
  - z = Zeit für einen vollständigen Zyklus

Berechnung der Prozessorauslastung

- Zyklus: Zeit für Rechnen, Verdrängen, Nachladen

  $z = b + (2*s) / r$

- Annahmen:
  - die Wartezeit w für E/A Operationen wird vollständig zur Bearbeitung anderer Prozesse genutzt.
    - w beeinflusst die Auslastung des Prozessors nicht
  - die Zeit (2*s) / r für Verdrängen und Nachladen kann nicht für die Prozessbearbeitung genutzt werden
    - während Verdrängen und Nachladen gibt es keinen rechenbereiten Prozess

Berechnung der Prozessorauslastung

$$L_p = \frac{b}{b + \frac{2 \cdot s}{r}} = \frac{1}{1 + \frac{2 \cdot s}{b \cdot r}}$$

$$s = 0{,}5\text{MByte}, r = 5\text{MByte/sec}, b = 0{,}5\text{sec}$$

$$L_p \approx 0{,}7$$

$$s = 0{,}5\text{MByte}, r = 5\text{MByte/sec}, b = 0{,}1\text{sec}$$

$$L_p = 1/3$$

Berechnung der Prozessorauslastung

- Fazit
  - die Prozessorauslastung sinkt mit kürzeren processor bursts

Swapping in mehreren festen Laufbereichen

- Bei mehreren festen Laufbereichen kann das Auswechseln zeitlich mit dem Rechnen von Prozessen überlappen.

- Prozessorauslastung $L_p(n)$:
  - n Laufbereiche müssen im Speicher anwesend sein
  - Speicher muss mindestens n+1 Laufbereiche groß sein
    - ein Laufbereich wird für das Auswechseln der Adressräume benötigt
  - Um mit den processor bursts Schritt zu halten darf ein Swapping-Vorgang nicht länger als ein processor burst b dauern.
  - Es gilt damit:

    $2s/r \leq b \rightarrow r_{min} = 2s/b$

Beispiel: Mindestdurchsatz

- s = 0,5 MByte, b = 0,5 sec: r_{min} = 2 MByte/sec.
- s = 0,5 MByte, b = 0,1 sec: r_{min} = 10 MByte/sec.

- Vergleich: Durchsatz einer Festplatte: 5 MByte/sec

- Fazit
  - Bei der Dimensionierung muss speziell auch der Durchsatz der Festplatte (Sekundärspeicher) betrachtet werden.
  - genauere Dimensionierung setzt genaue Kenntnis der Prozesse voraus, in der Praxis unrealistisch
- Pragmatische Lösung:
  - ein Laufbereich ist für lang laufende Prozesse reserviert
  - damit wird sichergestellt, dass für den Prozessor höchstwahrscheinlich ein rechenwilliger Prozess bereitsteht.

Swapping in dynamisch angelegten Laufbereichen

- Bei schwankender Anzahl von Prozessen und unterschiedlich großen Laufbereichen ist die Verwendung fester Laufbereiche ineffizient.
  - Adressräume erhalten einen Laufbereich gleicher Größe
- Dynamische Laufbereiche werden bei Prozesswechsel verdrängt oder Terminierung gelöscht.
  - Die Allokation von Laufbereichen im Speicher erfolgt über dynamische Segmentierung.
  - Dynamische Laufbereiche erzeugen externe Fragmentierung und Zerstückelung des Speichers.

Vermeidung von Fragmentierung

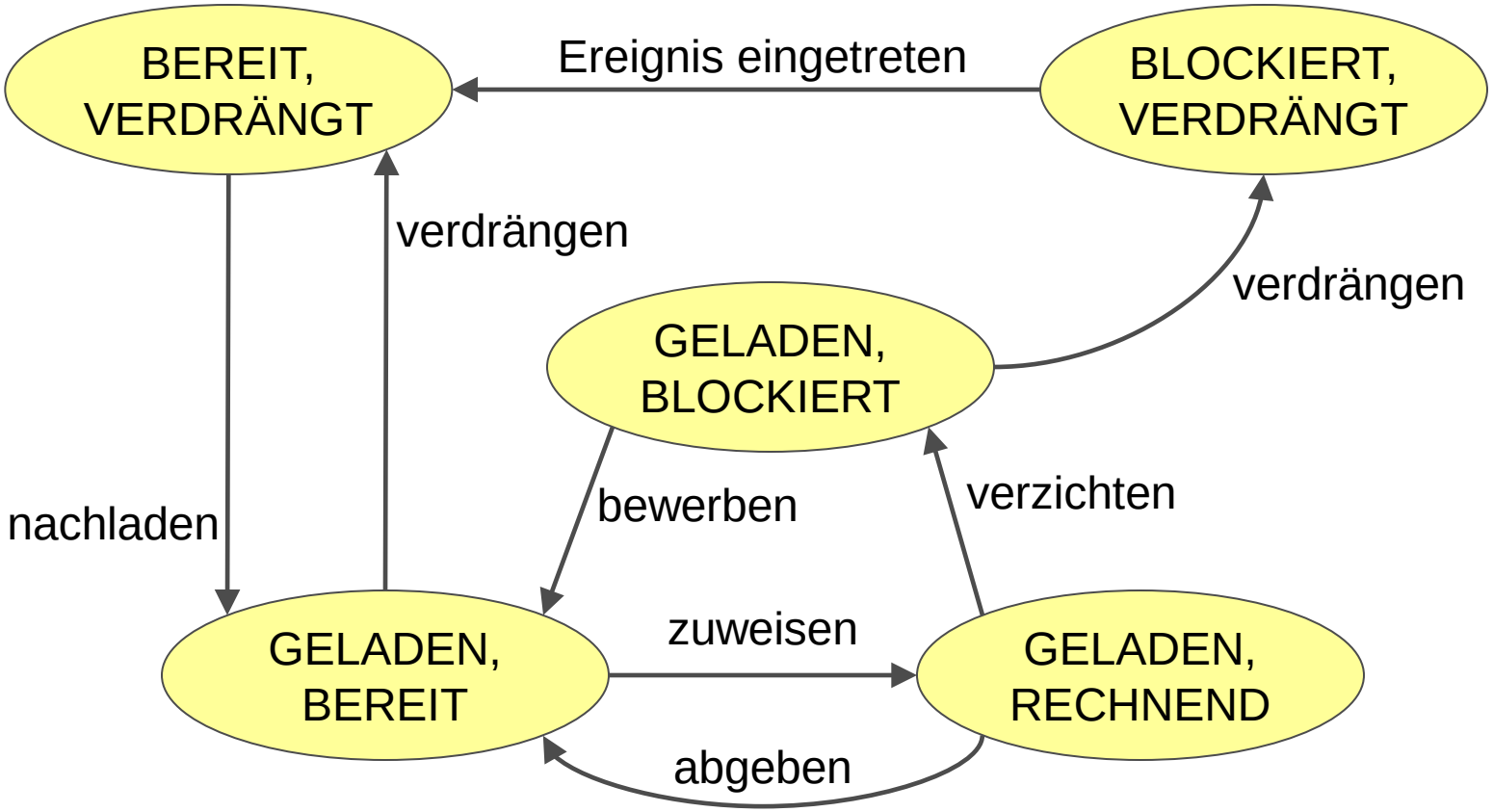
- Idee:
  - Laufbereiche müssen beim Nachladen an einer anderen Stelle platzierbar sein.
  - Dafür muss das Programm als dynamisch relokierbarer Code vorliegen:
    - alle Adressen im Code sind relativ
    - in der Praxis kaum möglich, Fragmentierung ist damit nicht vermeidbar
- Fazit
  - Fragmentierung kann verantwortlich dafür sein, dass rechenwillige Prozesse nicht nachgeladen werden können.
    - Effekt kann auch auftreten, wenn freier Speicher in der erforderlichen Größe verfügbar ist.

Implementierung für Swapping

- Betriebssystemkern muss verwalten, welche Prozesse im Speicher liegen oder verdrängt sind.
  - Der process control block (PCB) speichert dazu den Ort des Prozesses.
  - Für ausgelagerte Prozesse muss zusätzlich vom Kern gespeichert werden, ob sie fortsetzbar sind oder in einer Blockade ausgelagert wurden.

- Swapping erfordert neue Prozess-Zustände
  - GELADEN
    - Prozesse, die in einem Laufbereich im Speicher geladen sind
  - VERDRÄNGT
    - ausgelagerte Prozesse, die blockiert sind oder sich um Zuteilung bewerben

Erweitertes Prozess-Zustandsmodell



Implementierung des Speichermultiplexers

- Für das Multiplexen des Speichers werden ein Laufbereichsscheduler und ein Eventhandler benötigt.

- Eventhandler empfängt Ereignisse und gibt sie an den Laufbereichsscheduler weiter:
  - Zeitscheiben-Ende
  - Start einer lang laufenden Ein/Ausgabe
  - Ende einer Ein/Ausgabe
  - Terminierung

Implementierung des Speichermultiplexers

- Funktionsweise: Laufbereichsscheduler

1. suche einen verdrängten Prozess P_x
2. suche einen freien Laufbereich für P_x
3. lade P_x
4. erstmaliges Laden: PCB initialisieren
5. Zustand für P_x: GELADEN, BEREIT
6. weiter zu 1.

Implementierung des Speichermultiplexers

- Funktionsweise: Prozesserzeugung
  1. PCB anlegen
  2. Swap-Bereich auf Festplatte allokieren
  3. Code auf Swap-Bereich auslagern
  4. Prozess in Zustand BEREIT, GELADEN setzen
  5. Prozess im Zustand BEREIT verdrängen
  6. Laufbereichsscheduler informieren

- Funktionsweise: Zeitscheibenende
  1. Prozess in Zustand BEREIT, GELADEN setzen

Implementierung des Speichermultiplexers

- Funktionsweise: Prozess im Zustand BEREIT verdrängen
  1. Laufbereich auf Swap-Bereich auslagern
  2. Prozess in Zustand BEREIT, VERDRÄNGT setzen
  3. Laufbereichsscheduler informieren

- Funktionsweise: Start einer langsamen Ein/Ausgabe
  1. Prozess auf Swap-Bereich auslagern
  2. Prozess in Zustand BLOCKIERT, VERDRÄNGT setzen
  3. Laufbereich freigeben
  4. Laufbereichsscheduler informieren

Implementierung des Speichermultiplexers

- Funktionsweise: Terminierung
  1. Laufbereich freigeben
  2. Swap-Bereich auf der Festplatte freigeben
  3. PCB freigeben
  4. Laufbereichsscheduler informieren

- Ende einer Ein/Ausgabe
  1. Prozess in Zustand BEREIT, VERDRÄNGT setzen
  2. Laufbereichsscheduler informieren

- Bemerkung: Allokation des Swap-Bereichs muss nicht zwingend bei der Erzeugung ablaufen.

Virtuelle Adressierung

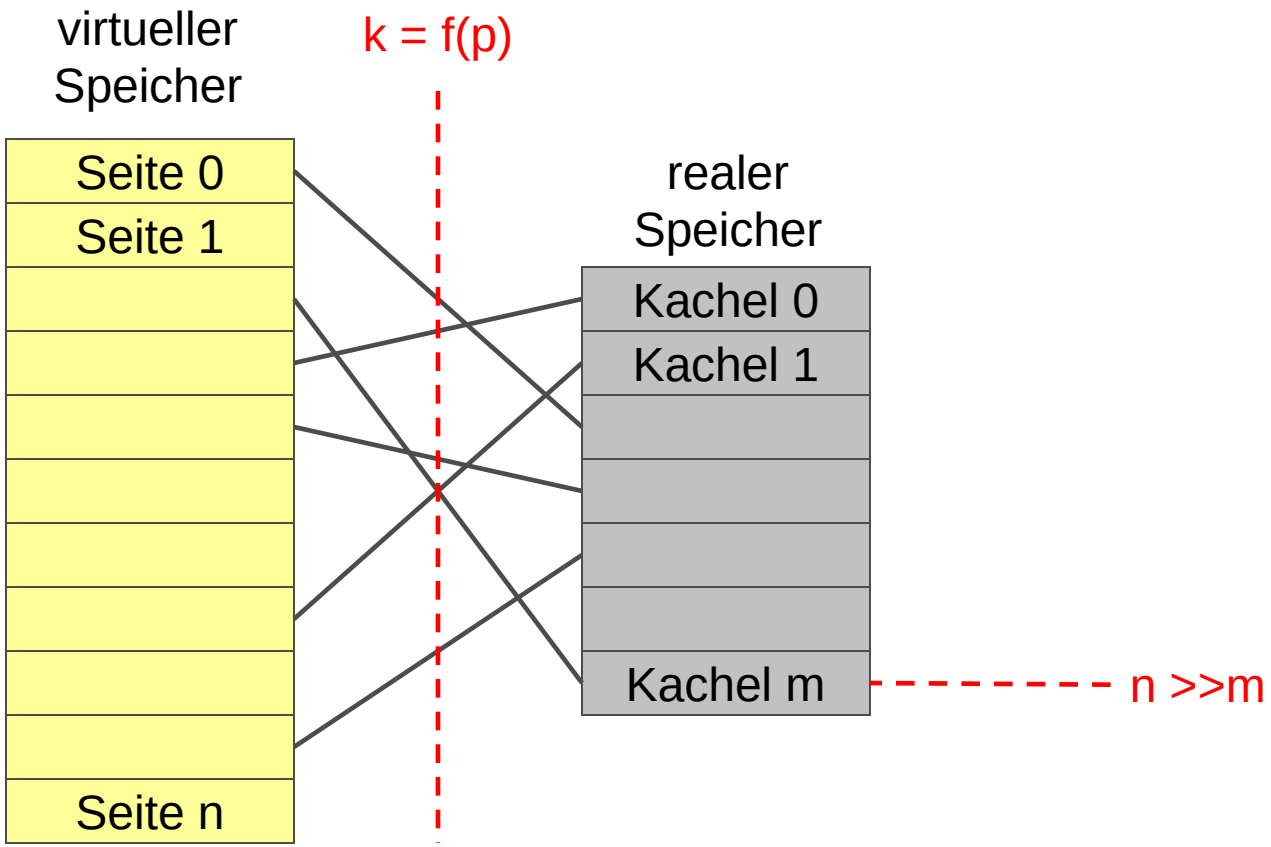
- Virtuelle Adressierung wird heute von den meisten Betriebssystemen unterstützt.
  - früher: teure Mainframe-Systeme
  - heute: auch Standard für PC-Systeme

- Eigenschaften virtueller Adressierung
  - Transparenz für Anwender
  - dynamische Verdrängung nicht benötigter Abschnitte eines Adressraums
  - dynamisches Nachladen benötigter Bereiche eines Adressraums
  - gute Speicherauslastung durch Vermeidung von Fragmentierung
  - volle Relokierbarkeit verdrängter Bereiche von Adressräumen

Prinzip der virtuellen Adressierung

- Der Prozessor nutzt physikalisch nicht existente Adressen (virtuell), die auf reale Adressen des Speichers abgebildet werden.
- Der virtuelle Speicher wird in Seiten unterteilt. Eine Adresse a des virtuellen Speichers wird aus 2 Angaben erzeugt:
  - Adresse a = (p,d) mit
  - p = Seitenadresse (engl. page) und
  - d = relative Adresse innerhalb der Seite (engl. displacement)
- Die Abbildung virtueller Adressen auf reale Adressen erfolgt seitenweise.
  - Seiten p werden auf Kacheln k des Speichers mit Abbildungsfunktion f abgebildet: k = f(p)
  - In der Praxis wird f durch eine Seitentabelle realisiert.

Prinzip der virtuellen Adressierung

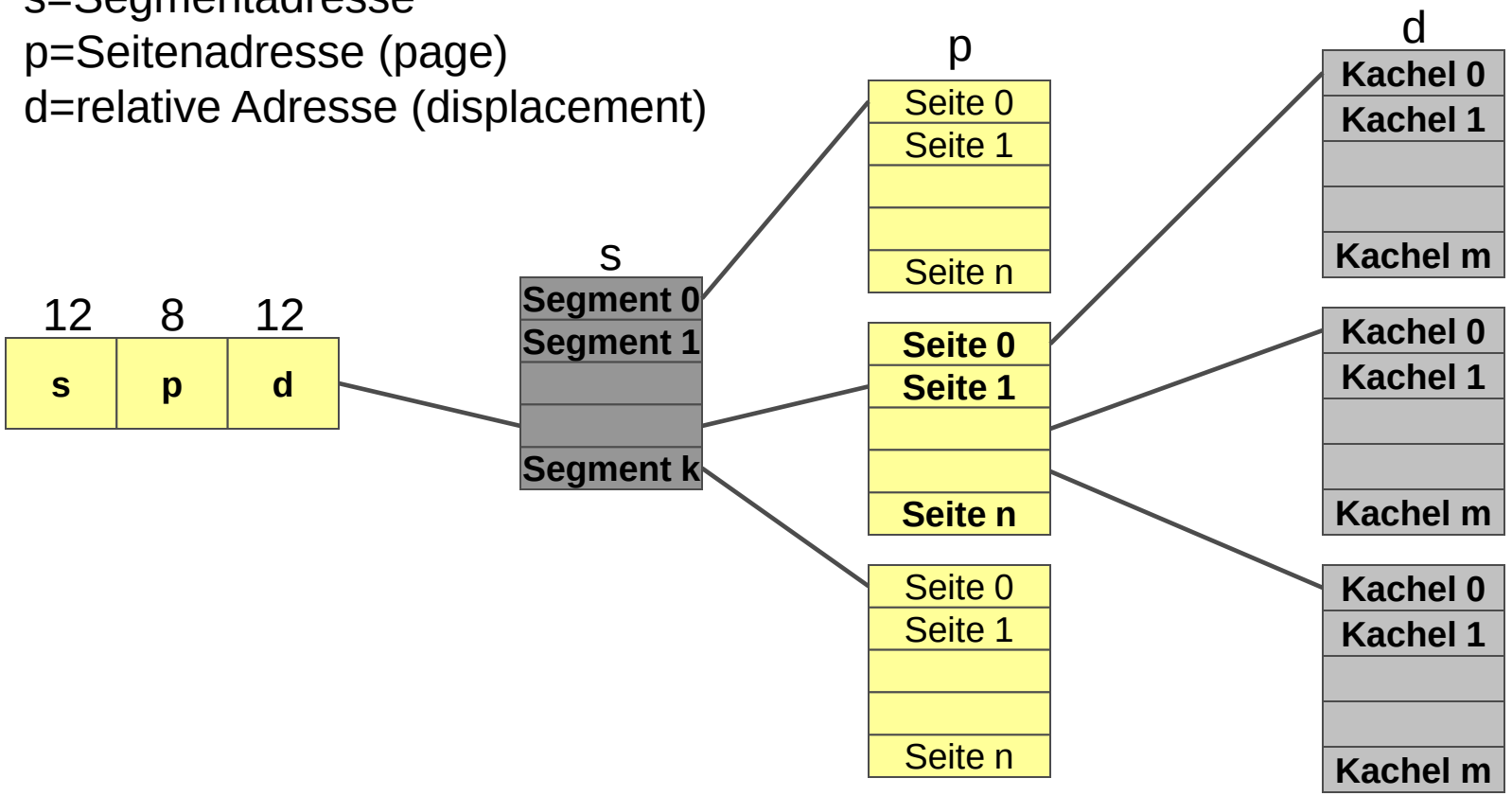


Verwaltung virtueller Adressen: Memory Management Unit (MMU)

- Eine MMU ist ein kleiner Prozessor, der die Adressabbildung vornimmt.
- Die MMU nutzt ein Register, dass auf den Anfang der im Speicher befindlichen Seitentabelle zeigt.
  - Register wird bei jedem Prozesswechsel umgeladen.
- Die Verwaltung von Seitentabellen kann einen hohen Speicheraufwand erfordern.
  - tatsächliche Belegung der Seitentabelle ist relativ gering.
- Ansatz zur Optimierung
  - mehrstufige Adressabbildung
  - eine Stufe indiziert die Seitentabelle

Beispiel: 3-stufige Abbildung

s=Segmentadresse
p=Seitenadresse (page)
d=relative Adresse (displacement)



Reduktion der Seitenfehlerwahrscheinlichkeit

- Das Verdrängen oder Ersetzen einer Seite ist der zeitaufwändigste Teil des Seitenzugriffs.
  - Ein Seitenfehler tritt auf, wenn eine adressierte Seite nicht im Speicher verfügbar ist.

- Ansatz zur Optimierung: Reduzierung der Seitenfehlerwahrscheinlichkeit
  - Ausnutzen von Lokalität: Nachladen einer Mindestzahl von Kacheln
  - Anwenden eines guten Seitenersetzungsalgorithmus (engl. paging)

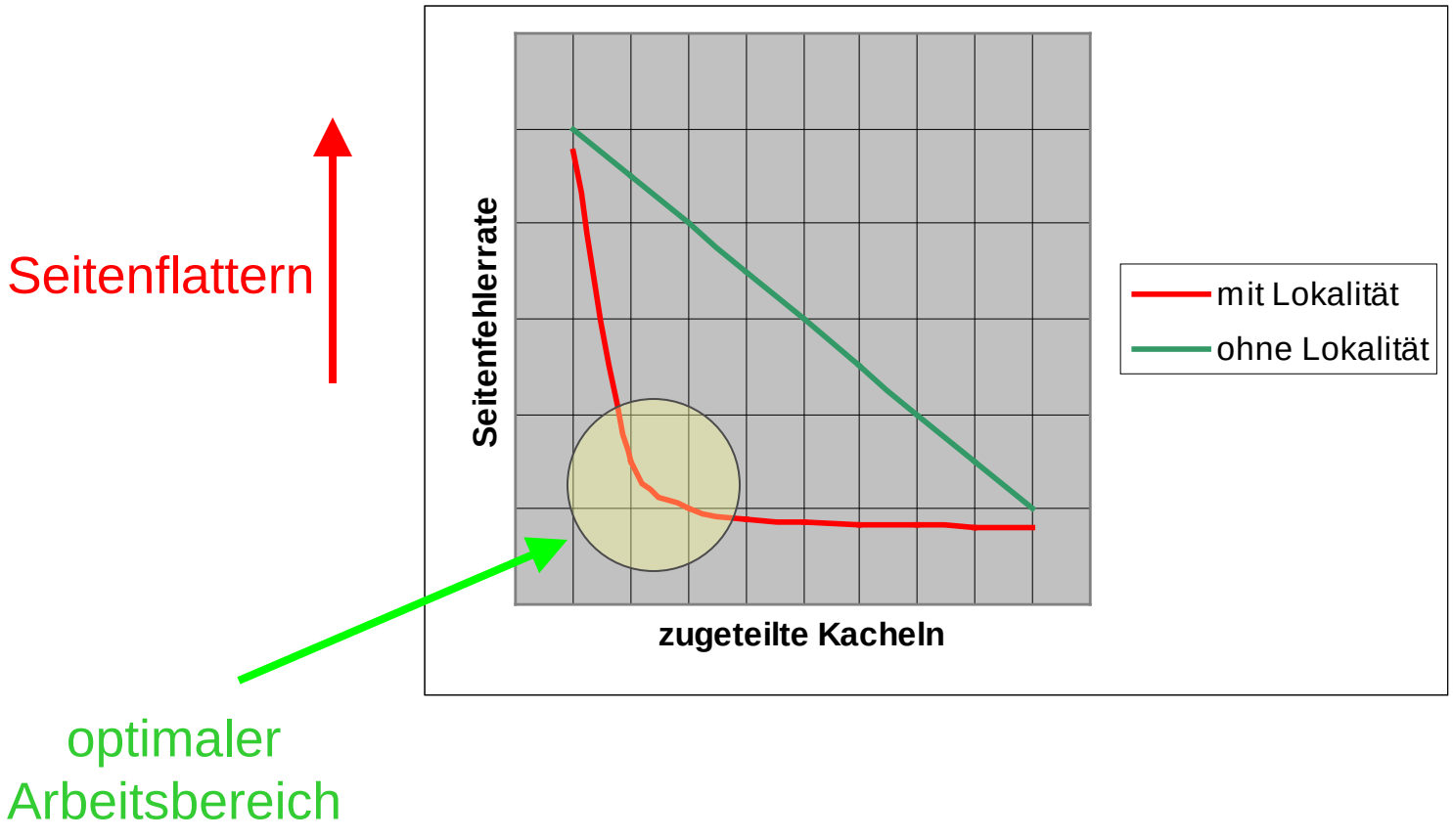
Kachelzuteilungsverfahren

- Ausnutzen der Lokalität reduziert die Seitenfehler-wahrscheinlichkeit.

- Lokalität protokollieren: Working Set
  - in periodischen Zeitabständen alle von einem Prozess referenzierten Seiten notieren
  - Alternative: messen der Seitenfehlerrate
- Locality Set
  - Working Set, dass die Lokalität gut abbildet

- Seitenflattern: System ist nur noch mit Ein- und Auslagern von Seiten beschäftigt
  - tritt auf, wenn die Seitenfehlerrate sehr hoch ist
  - kommt einem Zusammenbruch des Systems gleich!

Kachelzuteilungsverfahren



Seitenersetzungsverfahren (engl. paging)

- Seitenersetzung =

  Seitenverdrängung + Seitennachschub

- Seitennachschubverfahren
  - regelt, welche Seiten in freie Kachel des Speichers nachgeladen werden
  - Unterscheidung in Prepaging und Demand-Paging

Seitennachschubverfahren

- Prepaging
  - Laden erfolgt vor dem erstmaligen Zugriff, z.B. durch eine initiale Menge
  - Idee: Seitenfehler sollen gar nicht erst auftreten
- Demand-Paging
  - Laden erfolgt erst bei Zugriff

Seitenverdrängungsverfahren

- Seitenverdrängung regelt, welche Seite zu einem Zeitpunkt verdrängt wird
  - üblicherweise gekoppelt mit Demand-Paging
- Je schlechter die Seitenverdrängung, desto höher die Seitenfehlerrate.
- typische Seitenverdrängungsverfahren
  - optimale Verdrängung
  - Zufallswahl
  - FIFO (first in-first out)
  - LRU (least recently used)
  - Second Chance Algorithmus
  - LFU (least frequently used)

Seitenverdrängungsverfahren

- optimale Verdrängung
  - es wird die Seite verdrängt, die am längsten nicht zugegriffen wurde
  - als optimal beweisbar, aber in der Praxis nicht implementierbar
  - eignet sich gut für post-mortem Vergleiche
- Zufallswahl
  - zu verdrängende Seite wird zufällig gewählt
  - Verfahren ist nur bei gleich verteilten Zugriffen geeignet

Seitenverdrängungsverfahren

- FIFO (first in first out)
  - verdrängt die Seite mit der längsten Verweildauer
  - Implementierung
    - über Zeitstempel oder
    - mit einer FIFO Liste der Referenzen
  - Bewertung
    - besser als Zufallswahl
    - Referenzhäufigkeit wird nicht berücksichtigt

Seitenverdrängungsverfahren

- LRU (least recently used)
  - verdrängt die Seite, die am längsten nicht referenziert wurde
  - Implementierung
    - Seitenzähler zählt Zeitintervalle
    - Seiten werden in einer LIFO (last in first out) Liste verwaltet
    - bei Seitenfehler wird die Seite mit dem höchsten Zählerwert verdrängt
  - Bewertung
    - LRU approximiert die optimale Verdrängung!

Selbstkontrolle

1. Erläutern Sie starre Segmentierung
   1. mit einer festen Größe und
   2. mit mehreren festen Größen.
2. Wie funktioniert die Speicherorganisation bei dynamischer Segmentierung?
3. Vergleichen Sie First Fit und Best Fit bei dynamischer Segmentierung.
4. Erläutern Sie das Buddy-Verfahren.
5. Was ist ein Adressraum?
6. Was ist Swapping?
7. Was ist ein Laufzeitbereich?
8. Wie wird die Prozessorauslastung bei Swapping für den Zyklus Rechnen, Verdrängen, Nachladen berechnet?

Selbstkontrolle

1. Erläutern Sie das erweiterte Prozess-Zustands-modell.
2. Wie funktioniert ein Speichermultiplexer?
3. Erläutern Sie das Prinzip der virtuellen Adressierung.
4. Wie funktioniert 3-stufige Adressierung?
5. Was ist Seitenflattern?
6. Warum approximiert LRU eine optimale Verdrängung? Stellen Sie eine Hypothese auf.

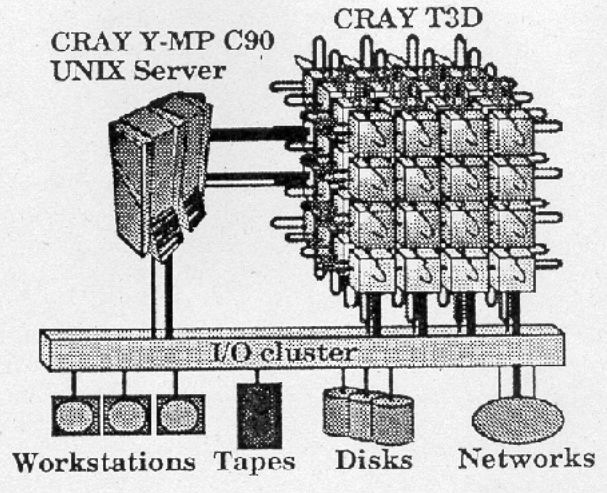
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Betriebssysteme
Unix



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Überblick

- Geschichte
- Struktur von Unix
- Dateisystem
  - Verzeichnisstruktur
  - Geräte
  - Festplattenorganisation
  - Zugriffsrechte
- Prozessorganisation
- Adressräume
- Kommandozeile

Geschichte

- 1965: MULTICS als Ergebnis eines Forschungsprojekts entstanden
  - MIT
  - Bell Laboratories
  - AT&T
  - General Electrics
- MULTICS vereinigte Eigenschaften, die bis zu diesem Zeitpunkt als nicht vereinbar galten:
  - Mehrbenutzerbetrieb (engl. multi-user)
  - Dialogbetrieb (engl. time-sharing)
  - gemeinsame, kontrollierte Dateibenutzung (engl. file-sharing)
- Probleme von MULTICS
  - nur für Großrechner geeignet
  - sehr kompliziert

Geschichte

- AT&T entwickelte ein MULTICS-basiertes, einfacheres Betriebssystem: UNIX.
  - für damalige Kleinrechner: PDP-7 (1969) und PDP-11 (1971)
- Eigenschaften von UNIX
  - entwickelt in der dafür entwickelten Programmiersprache C
  - < 3% verbleibender Assembler-Code
- UNIX wurde Universitäten kostenfrei für Experimentierzwecke überlassen.
  - 1974: Berkeley University gründet eine Gruppe, die UNIX weiterentwickelt
    - BSD-UNIX
    - erhältlich gegen geringe Lizenzgebühr
  - 1984: > 100.000 Installationen

Geschichte:
Technische Daten: PDP-7 und PDP-11

- DEC = Digital Equipment Corporation
- PDP = programmed data processor

- DEC PDP-7
  - Einplatz-System
  - seit 1964 im Einsatz
  - 18-Bit System
  - Preis: 120.000 US $

- DEC PDP-11
  - Mehrplatz-System
  - 16 Bit-System, 3-15 MHz Prozessor, 10 MB Festplatte, max. 4MB Speicher
  - von 1970-1990 verkauft
  - verschiedene Modelle, z.B. PDP 11/34: 60.000 Stück verkauft
  - Preis: ca. 10.000 US $

Geschichte: DEC PDP-7

Geschichte: DEC PDP-11

Geschichte

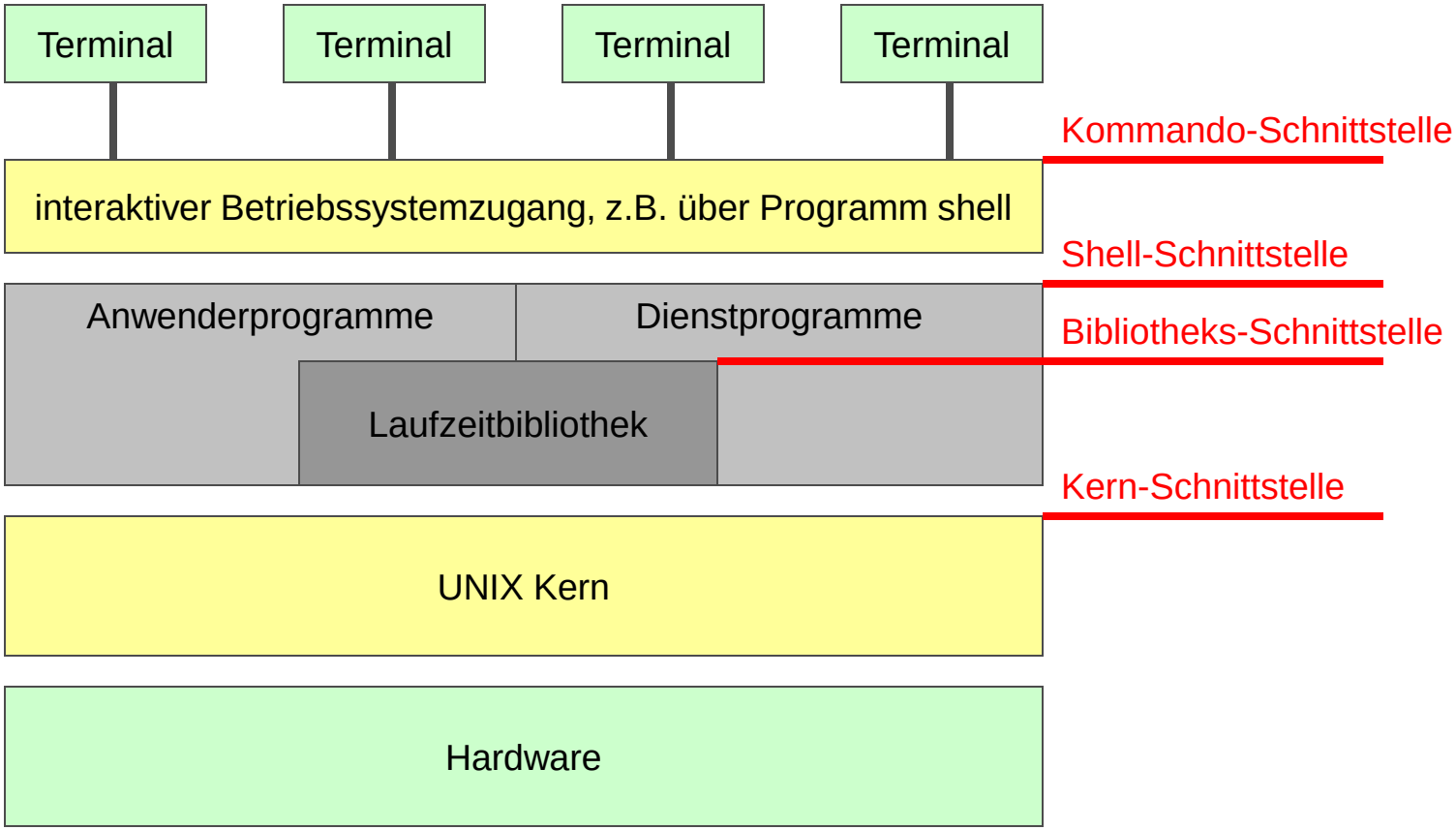
- 1983: AT&T vertreibt eigene UNIX-Version System V
  - starke Unterschiede zwischen System V und BSD
- 1988: Definition eines UNIX-Standards
  - POSIX 1003.2
  - besonders Sun unterstützte mit Solaris die Vereinigung der Fähigkeiten von System V und BSD
- heutige POSIX-kompatible UNIX
  - IBM AIX
  - HP HP-UX
  - Sun Solaris
  - SGI SGI UNIX
  - Linux

UNIX Systemarchitektur

- 3 Systemebenen
  - UNIX Kern (engl. kernel)
  - Bibliotheken
  - Standard-Dienste

- Basisdienste im UNIX Kern
  - Prozessverwaltung
  - Speicherverwaltung
  - Dateiverwaltung
  - Ein-/Ausgabe
  - Prozesskommunikation

UNIX Systemarchitektur



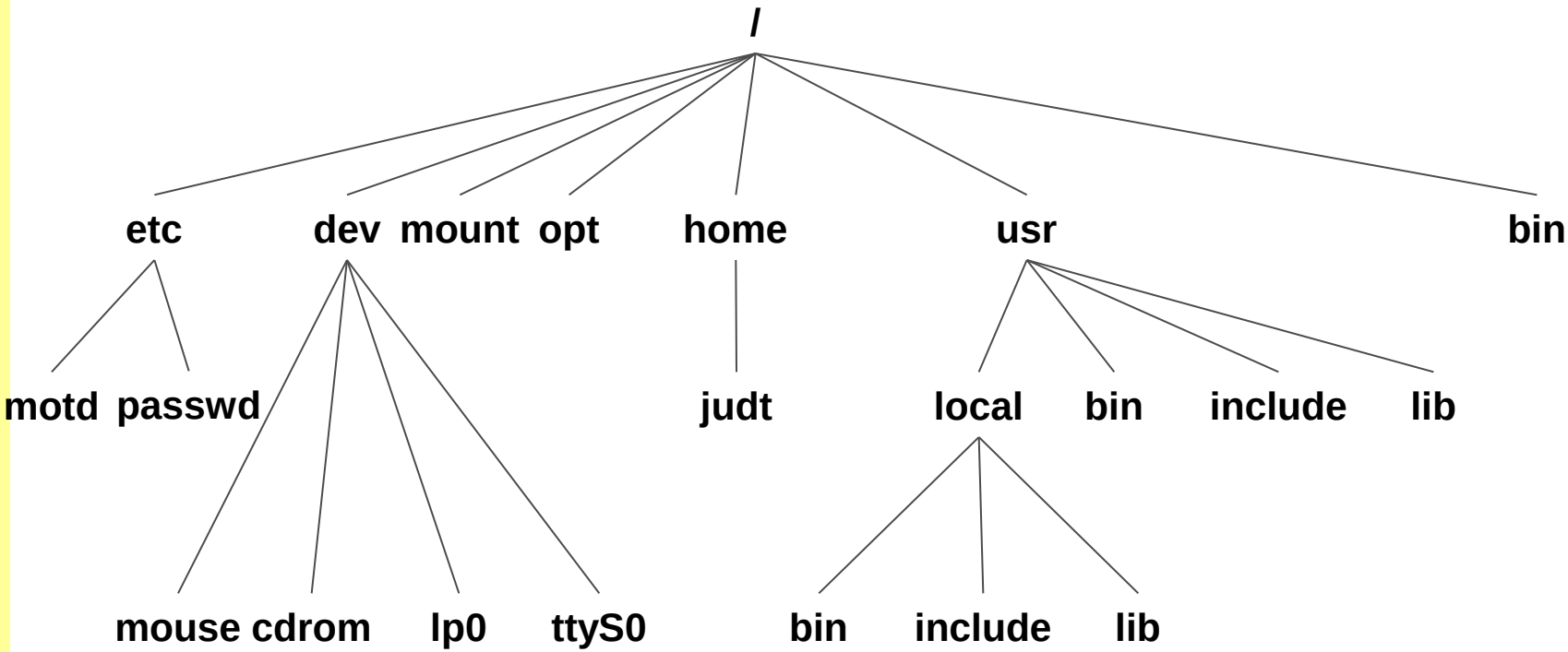
UNIX Systemarchitektur

- Größe einer typischen UNIX-Implementierung in C-Code
  - UNIX Kern
    - 350 KLOC
  - Sprachbibliotheken
    - 350 KLOC
    - z.B. C Laufzeitbibliotheken
  - Standard-Dienstprogramme
    - 330 KLOC
    - z.B. Compiler, Editoren, Shells

UNIX Dateisystem

- Das Dateisystem ist der zentrale Bestandteil eines UNIX-Systems.
  - Viele Teile des Betriebssystems werden auf Dateien zurückgeführt.
- Dateien werden als Bytestrom genutzt.
- Die Organisation von Dateien erfolgt durch Verzeichnisse, die als Baum organisiert sind.
  - Wurzel: „/“

UNIX Verzeichnisstruktur (Auswahl)



UNIX Verzeichnisstruktur

- „/“
  - Wurzel
- „/home“
  - Verzeichnis für Benutzerdaten
  - Unterverzeichnisse: Heimverzeichnisse der Anwender
    - in Variable $HOME
    - z.B. /home/judt
    - als „~“ adressierbar
      - „~“: Heimverzeichnis
      - „~judt“: Heimverzeichnis des Benutzers „judt“
- „/etc“, lat. et cetera
  - Konfigurationsdateien
    - z.B. passwd (Passworte), motd (engl. message of the day)
  - wichtige Shellskipte

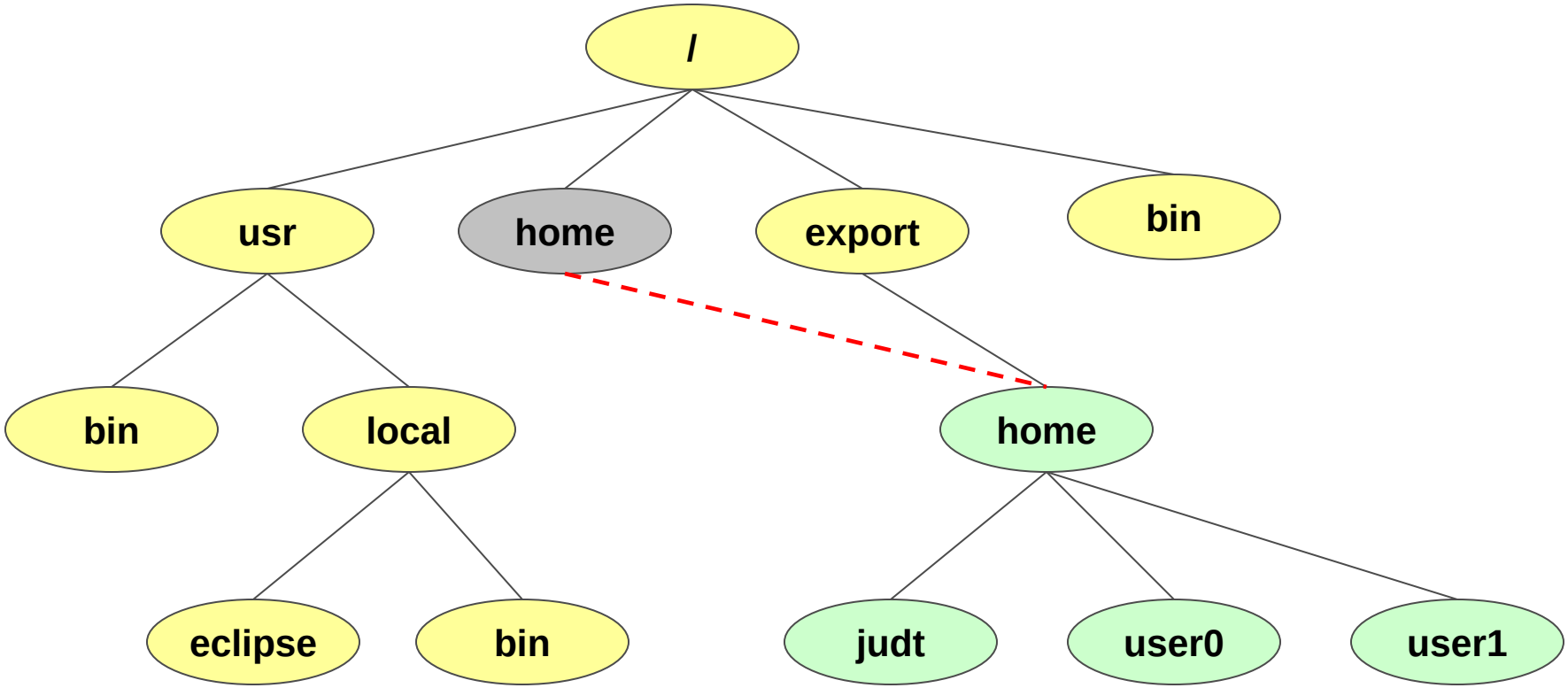
UNIX Verzeichnisstruktur

- „/bin“, engl. binary
  - Systemprogramme für den Administrator (Benutzer root)
- „/usr“, engl. user
  - Verzeichnis für Benutzerprogramme
  - z.B. Editoren, Compiler, Kompression, Archivierung
- „/usr/local“
  - Programme und Bibliotheken, die vom Administrator nachinstalliert wurden
- „/dev“, engl. devices
  - alle Geräte, als Dateien zugreifbar
- „/mount“
  - temporär eingebundene Verzeichnisse
  - z.B. von einer CDROM: „/mount/cdrom“

UNIX Verzeichnisstruktur

- Symbolische Verknüpfungen (engl. symbolic links)
  - Teilbäume können an beliebige Stellen im Verzeichnisbaum verknüpft werden.
- Inhalt eines Verzeichnisses
  - „.“: Repräsentant des aktuellen Verzeichnisses
  - „..“ Vaterverzeichnis
  - Dateien, Geräte, symbolische Verknüpfungen und Verzeichnisse werden gleich behandelt
- Mounts, NFS
  - Verzeichnisbäume können in Verzeichnisbäume integriert werden
    - Wechselmedien, z.B. CDROM
    - Verzeichnisse über mehrere Festplatten oder Partitionen
    - Einbinden von zentralen Verzeichnissen
      - z.B. Heimverzeichnisse auf Server unter „/export/home“

Beispiel:
Mount in /export/home,
Symbolische Verknüpfung /home → /export/home



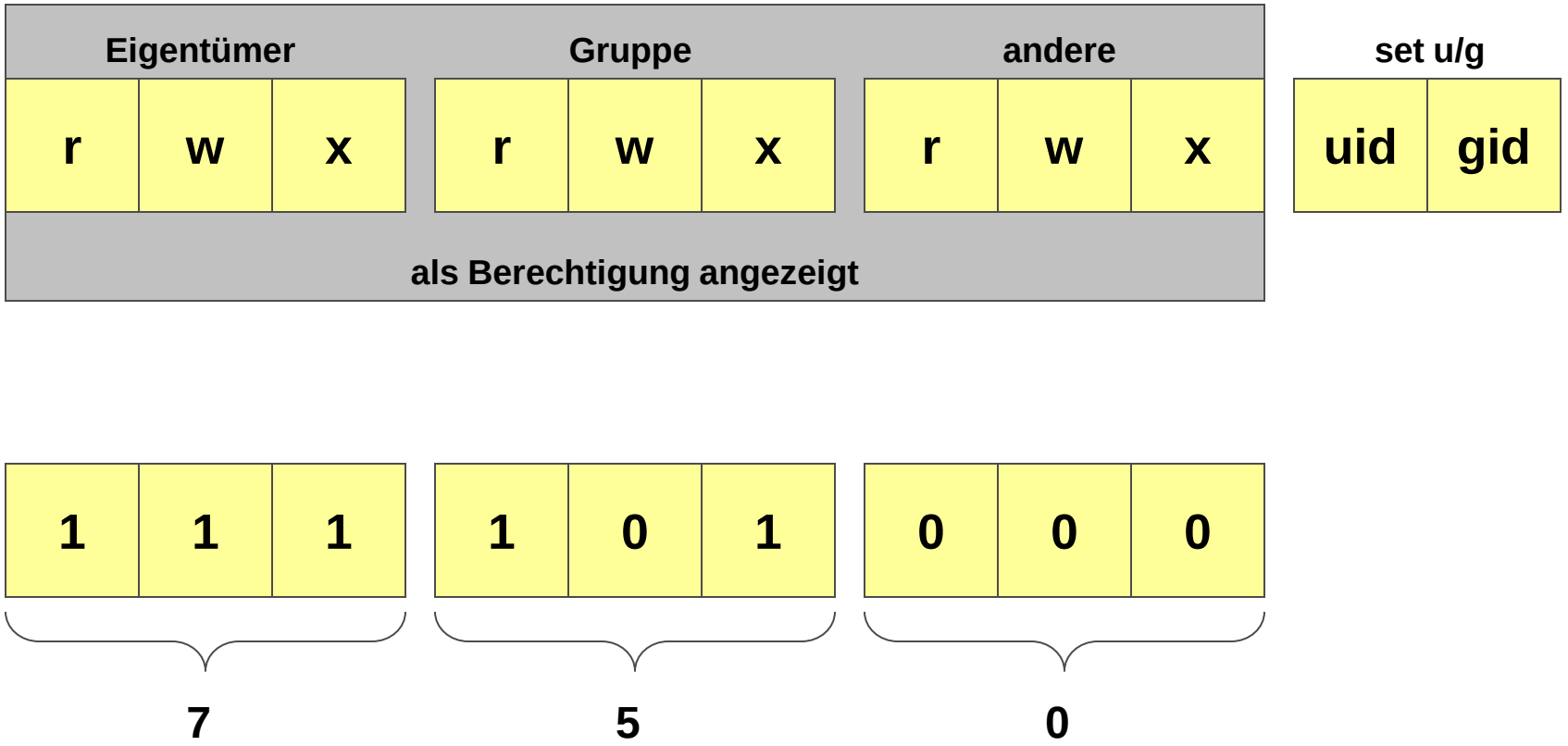
Berechtigungen: Schutz von Dateien und Programmen

- Dateien und Verzeichnisse werden durch 11 Bits eingestellt.
  - 3 Bits für den Eigentümer (engl. user)
  - 3 Bits für die Gruppe (engl. group)
  - 3 Bits für alle anderen (engl. others)
  - 2 Bits für die Übertragung von Rechten während der Programmausführung
- 3 Bits für Eigentümer, Gruppe, andere
  - read (r): Lesen erlaubt
  - write (w): Schreiben erlaubt
  - execute (x): Ausführen erlaubt

Berechtigungen: Schutz von Dateien und Programmen

- 2 Bits zur Übertragung von Rechten
  - set_user_id: erzwingt bei Programmausführung die Umschaltung zur Identität den Eigentümers
  - set_group_id: erzwingt bei Programmausführung die Umschaltung zur Gruppe den Eigentümers
  - Fremden Benutzern kann damit zur Programmausführung eine andere Berechtigung gegeben werden.

Berechtigungen: Schutz von Dateien und Programmen



Vergleich:
Zugriff auf Dateien und Verzeichnisse

Recht \ Typ	Datei	Verzeichnis
r	Datei lesen	Verzeichnis lesen
w	Datei schreiben	Dateien oder Verzeichnisse löschen oder hinzufügen
x	Datei ausführen	Wechseln in das Verzeichnis

Dateisystem auf der Festplatte

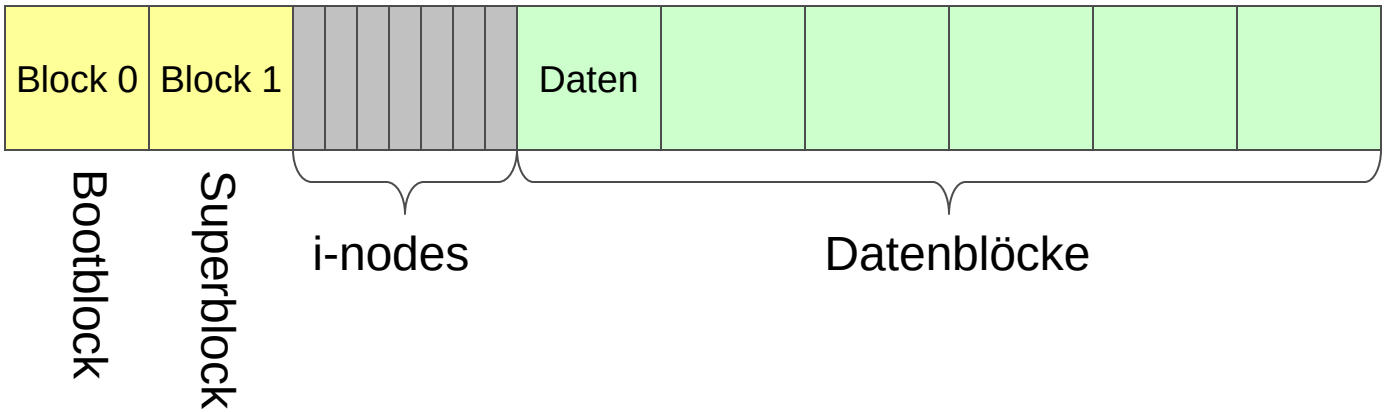
- Festplatte ein Partitionen mit verschiedenen Zwecken unterteilt, z.B.
  - /boot: Dateien, die zum Starten des Systems nötig sind
  - /home: Benutzerdateien
  - /tmp: temporäre Daten
  - /usr: Anwendungsprogramme
  - SWAP: Datenbereich zur Auslagerung von Speicherseiten

- Organisation einer Partition erfolgt in Blöcken
  - Block 0: Bootblock, Daten zum Starten des Systems
  - Block 1: Superblock, Informationen über die weitere Aufteilung der Partition
  - Block 2: Beschreibung der Dateikopfliste
  - Block 3…(n+3): Dateikopfliste (engl. i-nodes), organisiert Dateien und Verzeichnisse

Typische Verwaltung von Dateien und Verzeichnissen in i-nodes

- i-nodes sind typischerweise 64 Byte lang
- Block 2:
  - Anzahl der verfügbaren Blöcke
  - Beginn der Liste freier i-nodes
- i-node 0:
  - verwaltet defekte Blöcke
- i-node 1:
  - verweist auf Wurzelverzeichnis „/“
- Verzeichnis-Eintrag
  - 14 Byte für den Verzeichnisnamen
  - 2 Byte für Index des zugehörigen i-nodes in der Dateikopfliste
- Öffnen einer Datei erfolgt durch sukzessives Durchlaufen der i-nodes des Pfades, bis der zugehörige i-node gefunden ist.

Typische Partition eines UNIX-Systems



Spezielle Dateien unter /dev

- Unter „/dev“ (engl. devices) werden Geräte des Rechners als spezielle Dateien verwaltet, z.B.
  - Festplatte
  - Netzwerkkarte
  - Soundkarte
  - Terminals

- Der Zugriff auf Geräte erfolgt über Schreib- oder Leseoperationen auf Dateien.

Spezielle Dateien unter /dev

- Unterscheidung der Geräte
  - block special files
  - character special files
  - network special files
- Auf block special files kann blockweise auf beliebige Adressen geschrieben werden.
  - z.B. Festplatte, Grafikkarte
- Auf character special files wird über Zeichenströme zugegriffen
  - z.B. Maus, Drucker, Plotter, Terminal

Sockets

- Network special files sind Kommunikations-Endpunkte des Systems
  - Kommunikation wird über die Erzeugung von Sockets etabliert.

- 3 Socket-Typen:
  - zuverlässig, verbindungsorientiert, Bytestrom
  - zuverlässig, verbindungsorientiert, Paketstrom
    - z.B. TCP/IP
  - unzuverlässig, verbindungslos, Paketstrom
    - z.B. UDP

UNIX Prozesse

- Prozesse werden baumartig organisiert.
- Wurzel: Prozess Nr. 1
  - sog. Init-Prozess
  - vom Kern gestartet
  - läuft mit Administrator-Rechten (root, user id 1)
- Init-Prozess erzeugt eine Vielzahl von weiteren Prozessen.
  - In der Prozessliste (z.B. Kommando ps) wird für jeden Prozess auch der zugehörige Vater-Prozess angegeben.

UNIX Prozesse

- Erzeugen eines Prozesses: fork
  - Prozess wird dupliziert mit gleichem PCB
  - Unterscheidung von Vater- und Sohnprozess
    - fork liefert im Vaterprozess die ID des Kindes (pid > 0)
    - fork liefert im Kindprozess Null (pid = 0)
    - fork liefert im Fehlerfall eine negative ID (pid < 0): Kindprozess konnte nicht erzeugt werden

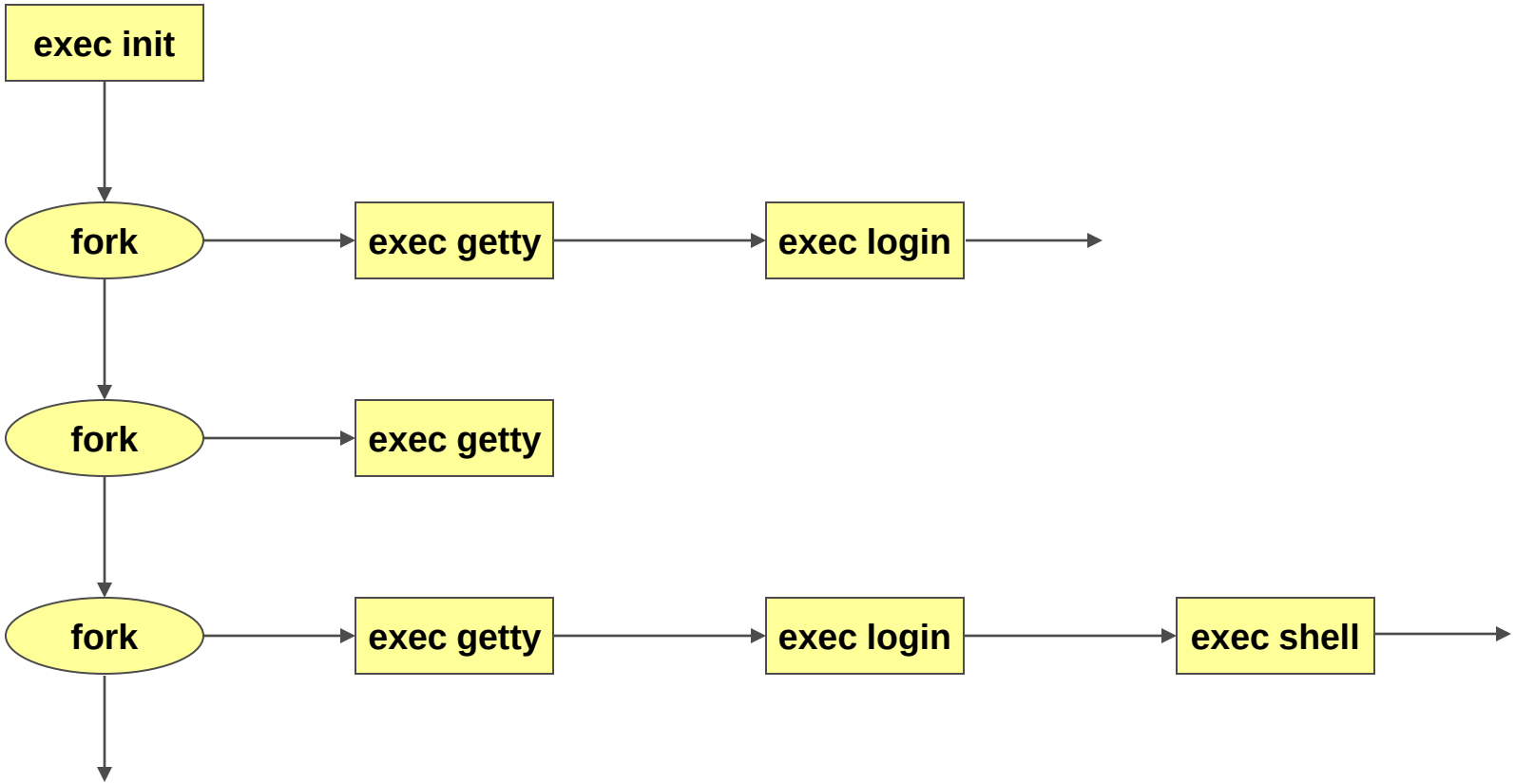
UNIX Prozesse

- Jedes UNIX-Kommando hat die Erzeugung eines Kind-Prozesses zur Folge.
  - auch durch exec-Aufruf aus einem Programm
  - Vater-Prozess kann auf das Ende seines Kindprozesses warten.

- Zombie-Prozess
  - Shell-Prozess, der beendet wurde und auf die Terminierung seiner Kind-Prozesse wartet.

- Pipes: Datenaustausch zwischen Prozessen
  - Ergebnisse von Prozessen können miteinander verknüpft werden.
  - Beispiel: „cat /var/log/messages | grep „xyz“ | more“

Beispiel: UNIX Prozessbaum



Traps unter UNIX: Signale

- Softwareseitige Unterbrechungen (engl. traps) werden als Signale (engl. signals) bezeichnet.
  - Signale werden zwischen zwei Prozessen ausgetauscht.
- Empfangender Prozess muss möglichst schnell in den Unterbrechungsmodus umschalten.
- 3 Möglichkeiten der Reaktion auf ein Signal
  1. Das Signal wird ignoriert.
  2. Das Signal wird durch eine Unterbrechungsbehandlung bearbeitet.
  3. Das Signal löst die Terminierung des Prozesses aus (Standardverhalten).
- Beispiel:
  - Signal 11: fehlerhafte Speicherallokation (engl. segmentation fault)

Selbstkontrolle

1. Erläutern Sie den Unterschied zwischen MULTICS, System V, BSD und POSIX 1003.2
2. Wie sind UNIX Dateisysteme organisiert?
3. Wie funktioniert ein Mount?
4. Wie wird unter UNIX auf Geräte zugegriffen?
5. Welche Aufgabe hat der Init-Prozess?
6. Wie funktioniert fork?

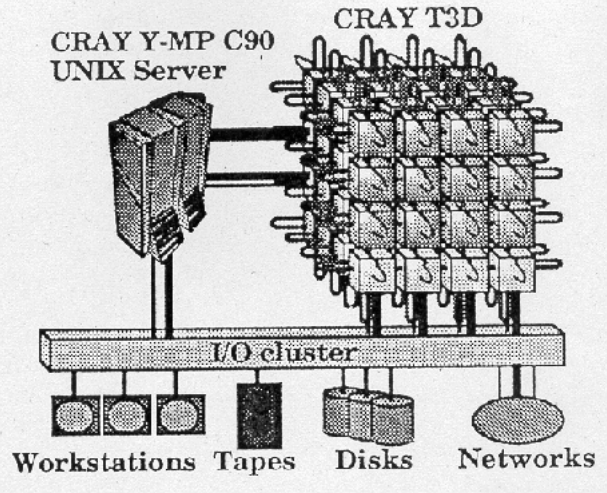
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

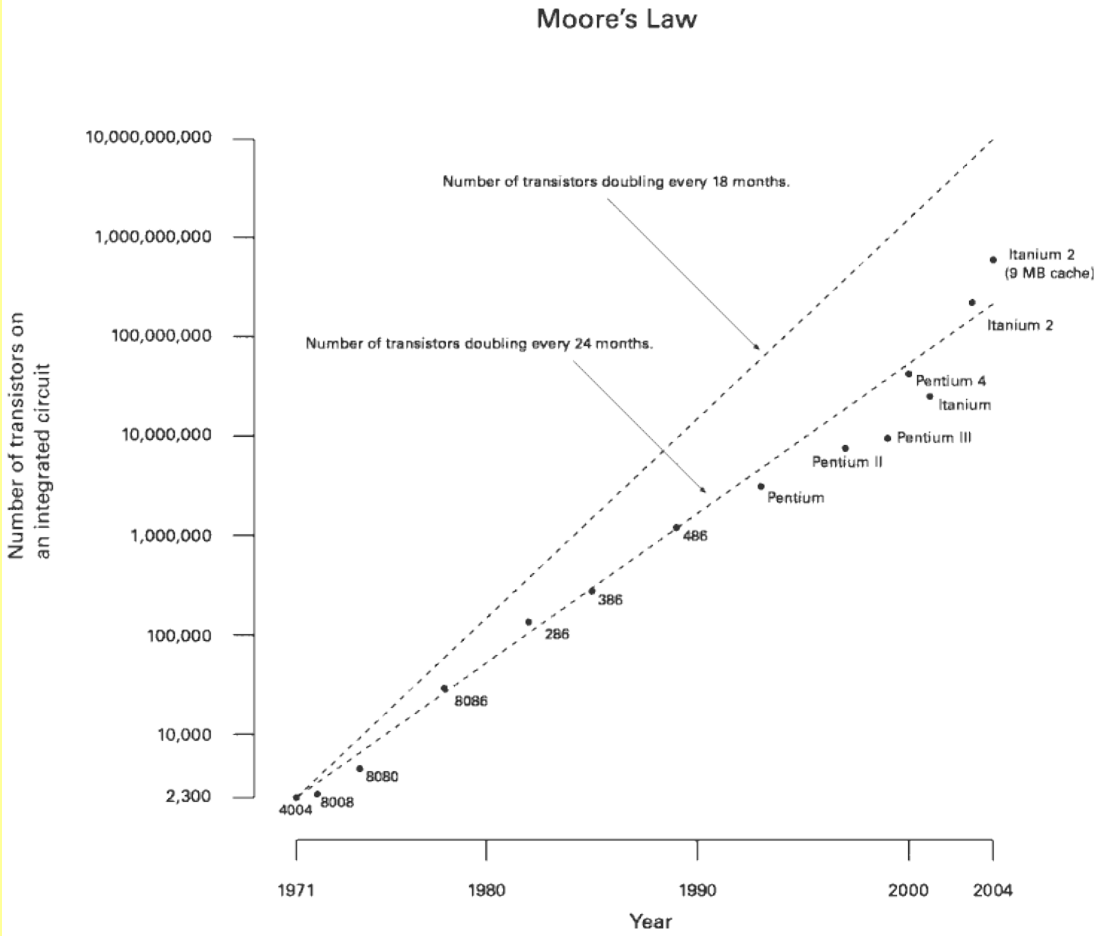
Betriebssysteme
Moderne Prozessorarchitekturen

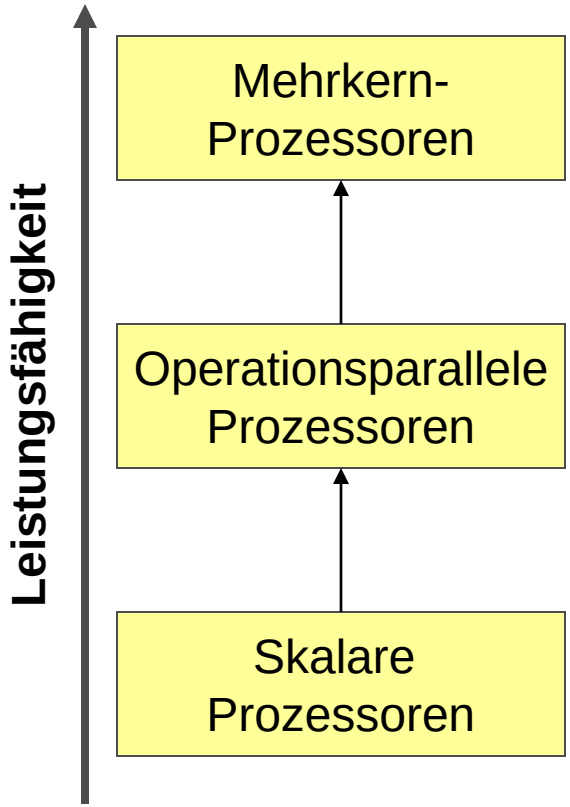


Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

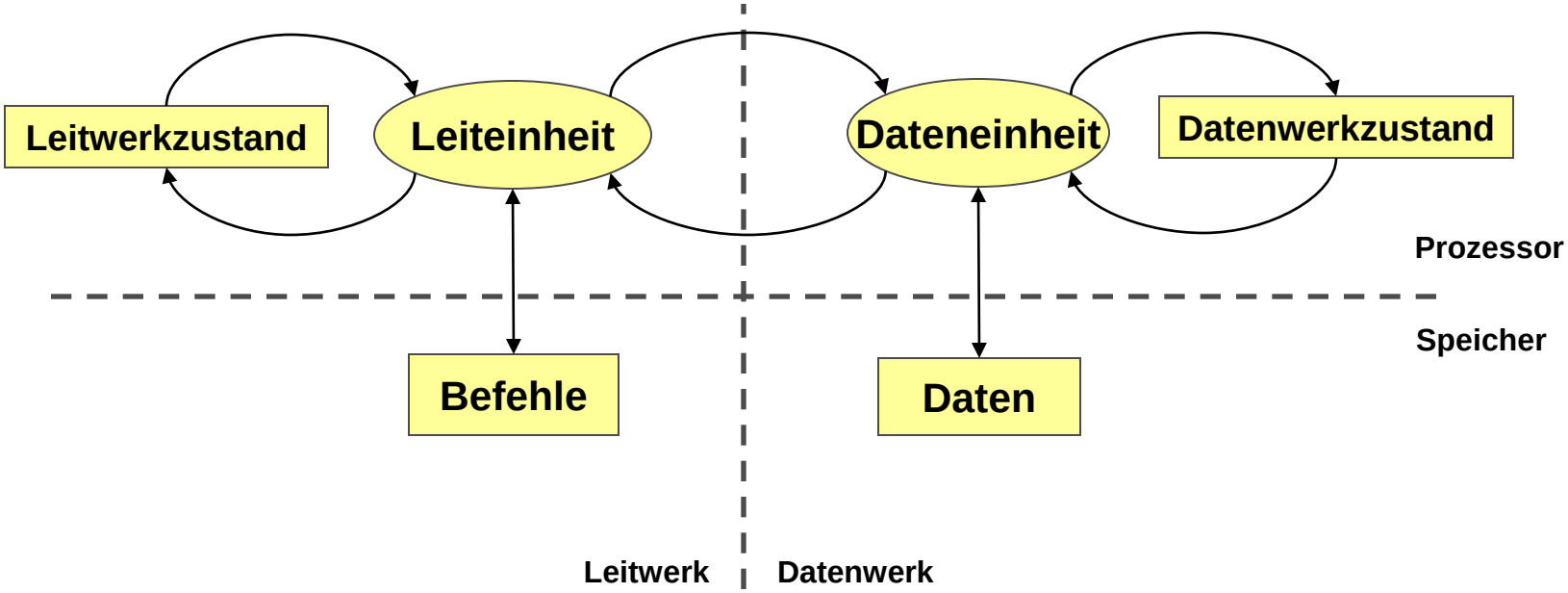
Prozessorarchitekturen vs. Moore‘s Gesetz





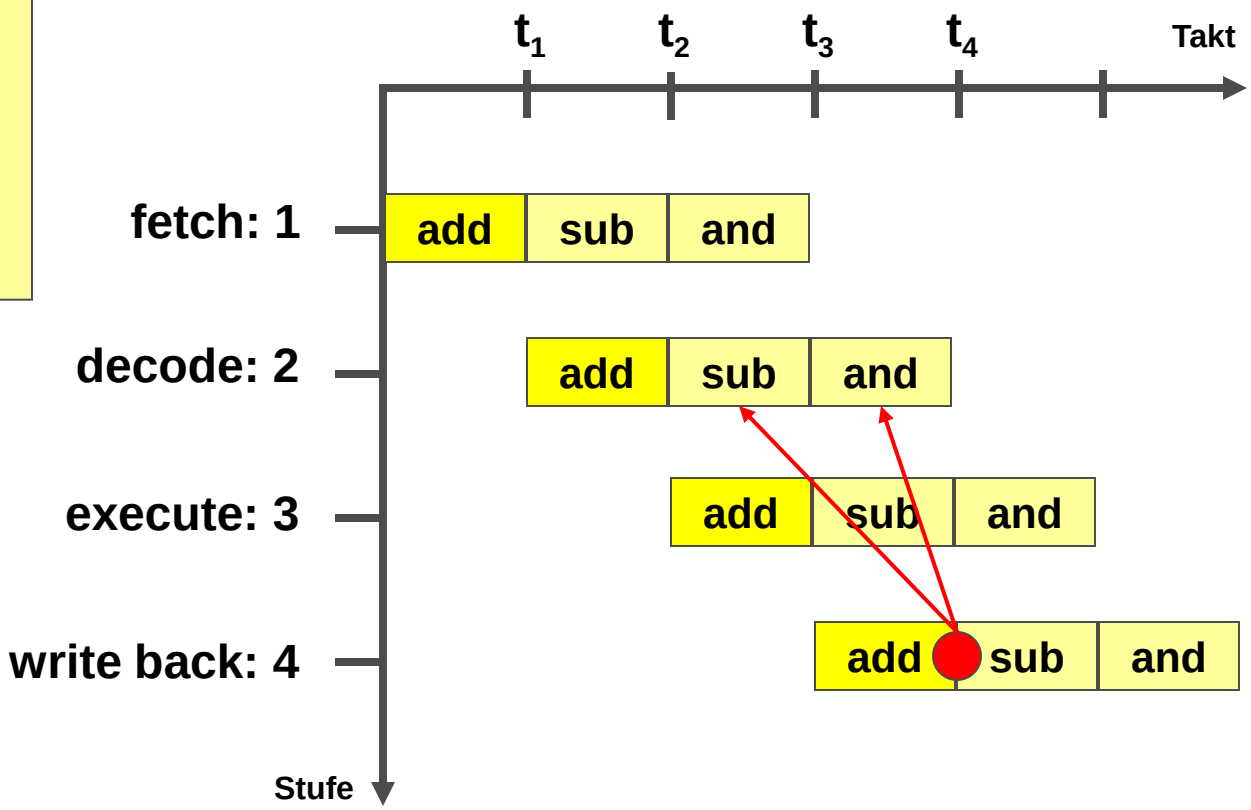
Skalare Prozessoren

- Sequentielle Verarbeitung von Programmen nach dem Kontrollflussprinzip
- Funktionale Beschreibung mit Registertransferschaltungen



Geschwindigkeitssteigerung durch Fließbandverarbeitung (Pipelining)

```
add r1, r2, r3
sub r3, r4, r1
and r5, r1, r5
```



Konfliktlösung bei Datenflussproblemen (Auswahl)

- Austauschen von Ergebnissen nach der execute-Phase (Bypassing)
  - Transfer über spezielles Austauschregister
- Sperren des Fließbandes, bis Daten vorliegen (Interlocking)
  - Einfügen von nop Befehlen.
  - Fließband läuft im schlechtesten Fall leer.
- Vorhersage von Ergebnissen für bedingte Programmverzweigungen (Branch Prediction)
  - verschiedene Strategien, z.B. Branch Caches, Predication

Operationsparallele und mehrfadige Verarbeitung (Multithreading)

- Bearbeiten von Operationen statt Befehlen
  - Maß für Operationsparallelität (issue): Zahl der Operationen pro Zeiteinheit, die den Befehlsdecoder maximal verlassen
- Parallelisierung von Programmen wird erreicht durch…
  - optimierenden Übersetzer
  - explizite Kennzeichnung durch Programmkonstrukte
- Parallelisierung von Operationen wird angewendet…
  - z.B. bei Mehrprozessorsystemen
  - z.B. Multimedia-Einheiten
    - z.B. SSE2-Einheit bei Pentium4 für Vektormathematik
- Multithreading: parallele Verarbeitung von Operationen durch Kontextwechsel
  - Prozessor und Ressourcen besser auslasten
  - verschiedene Strategien für Kontextwechsel

Technische Grenzen

- Leistungssteigerung mit bisherigen Prozessoren
  - Erweiterung der Befehlsätze, z.B. Intel MMX oder AMD 3DNow!
  - Erhöhung der Taktfrequenz
  - Verbesserte Strategien für Pipelining, Branch Prediction und Multithreading

- Größtes Problem:
  - Kaum handhabbare Abwärme bei Taktfrequenzen von ca. 4 GHz und mehr.
  - enorme Stromaufnahme (> 200 Watt)
- Auswirkung:
  - Ende 2005 stagnierte die Entwicklung von Einzelprozessoren.

Mehrkern-Prozessoren (Multi Core)

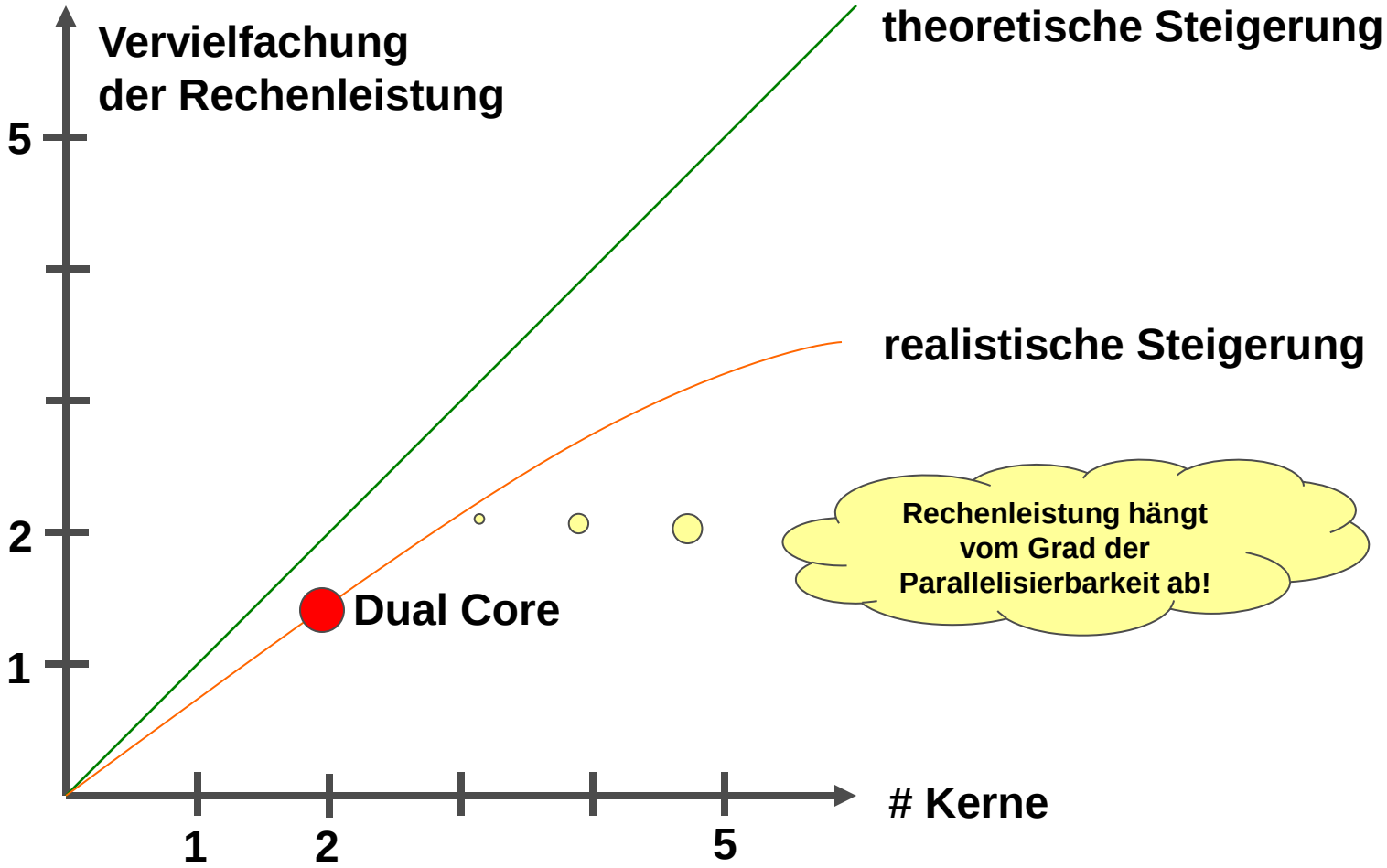
- Mehrkern-Architekturen sind heute zentraler Aspekt der Performanzverbesserung
  - Abwärme besser abführbar
  - Herstellungskosten nicht wesentlich höher
- Einfache Skalierung der Leistung durch Zahl der Kerne (n-Core)
  - Stand heutiger PC-Systeme: 2-Kern und 4-Kern Prozessoren
  - theoretisch: n Kerne = n-fache Leistung

- Betriebssystem und Programm müssen parallelisiert werden.
  - Programmfragmente als Threads auf mehreren Prozessoren
  - 2 Ansätze: Shared Memory Programmierung (SMP) und Message Passing Interface Programmierung (MPI)

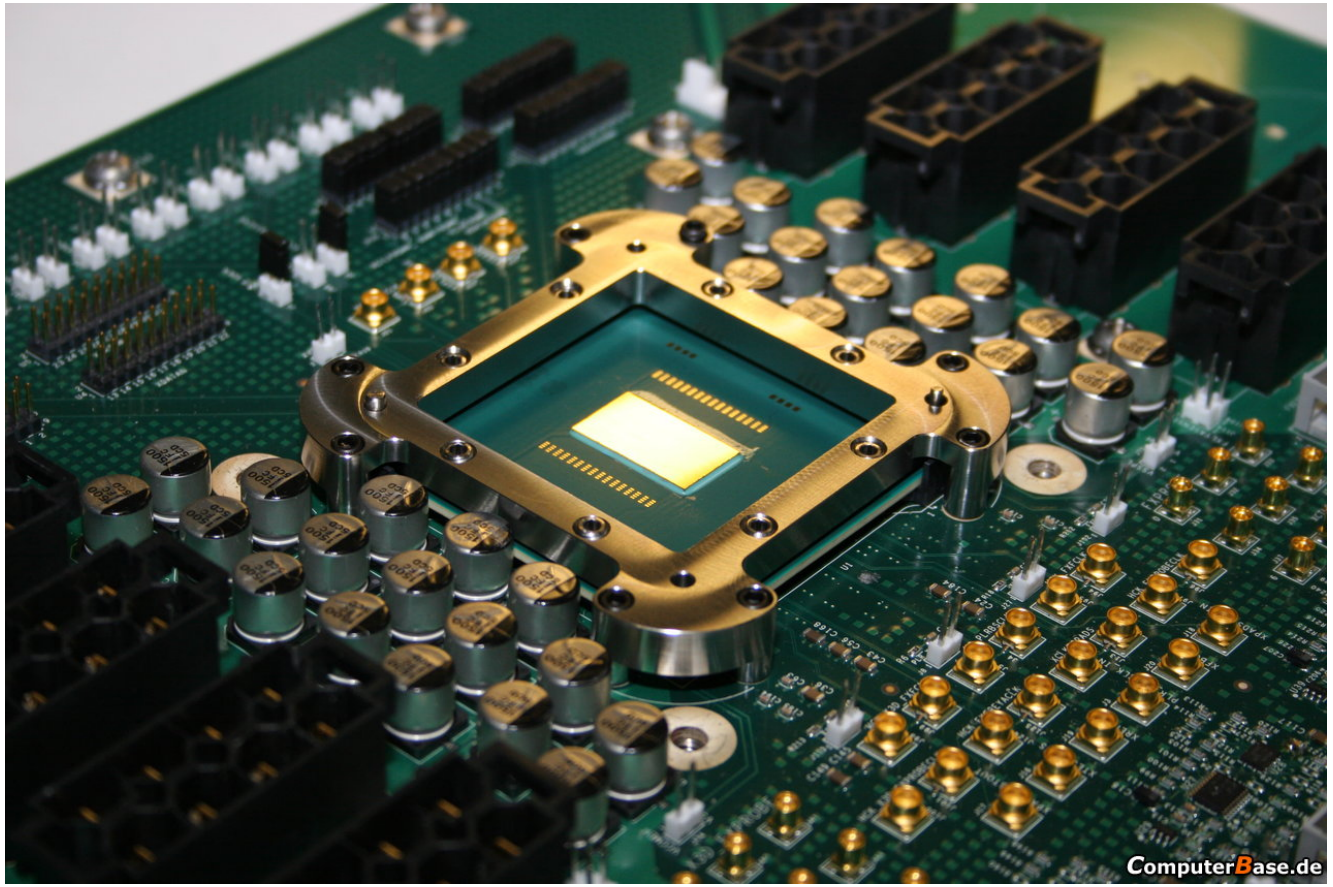
Beispiele für Mehrkern-Prozessoren

	2-Core	4-Core	8-Core und größer
Intel	Core 2 Duo, Pentium D, Itanium 2	Core 2 Extreme, Core 2 Quad, Xeon, Core i7	Xeon, Polaris (80 Kerne, experimentell)
IBM	POWER 5+, PowerPC 970MP	POWER 5+, POWER 6	64 Bit PowerPC, (z.B. Playstation 3), POWER 7 (2010)
AMD	Opteron, Athlon 64 X2, Turion 64 X2	Quad FX, Operon, Phenom X4, Phenom II X4	Opteron, Radeon, FireStream
Sun Microsystems	SPARC IV+	UltraSPARC T1	UltraSPARC T1, UltraSPARC T2, UltraSPARC T3 (16 Kerne)

n-Core != n-fache Rechenleistung



Beispiel: Intel Polaris (2007)
80 Kerne, 1 Tera Flop, 62 Watt



Auswirkungen von Mehrkern-Architekturen auf die Software-Entwicklung

- Viele Softwaresysteme bereits parallelisierbar
  - Datenbanken, Webserver, viele numerische Anwendungen
- Die meisten Programme müssen aber noch parallelisiert werden.
  - Betriebssystem für Mehrprozessorplattformen sind Stand der Technik
  - händische Lösung zu aufwändig, parallelisierende Compiler werden erforscht
- Weiterentwicklung von Programmiersprachen und Programmbibliotheken ist dringend erforderlich.
  - Parallelisierung soll implizit mit Hochsprachen erreicht werden!

Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de