Systeme I: Betriebssysteme

Wiederholung wichtiger Inhalte der Kapitel 3-7

Maren Bennewitz

Version 12.02.2013

Hinweis

- Diese Folien haben keinen Anspruch auf Vollständigkeit, was den Inhalt der Prüfung angeht
- Sie dienen nur dem Zweck der (kurzen) Wiederholung wichtiger Inhalte, die Sie gelernt haben sollten
- Insbesondere kann der Inhalt von Kapitel 8 selbstverständlich auch in der Prüfung abgefragt werden

Dateisysteme

Dateisysteme

- Dateisystem = Betriebssystemteil, der sich mit Dateien befasst
- Externe Sicht: Vom Dateisystem angebotene Funktionalität
- Interne Sicht: Implementierung von Dateisystemen

Dateiattribute

- Informationen über eine Datei, die das Betriebssystem speichert
- Beispiele:
  - Entstehungszeitpunkt (Datum, Uhrzeit)
  - Dateigröße
  - Zugriffsrechte

Zugriffsrechte

- Sicht des Benutzers (Bsp. Unix / Linux):

```
$ ls –l
drwxr-xr-x 6 maren users 238 Oct 18 12:13 systeme-slides
drwxr-x–- 1 maren users 204 Oct  2 11:10 systeme-uebungen
-rw------- 1 maren users 29696 Feb 29  2012 zeitplan.xls
```

- Bedeutung der Felder:
  - Typ des Eintrags: Datei, Verzeichnis
  - Rechte: Besitzer, Gruppenbesitzer, alle anderen
  - Anzahl Einträge (Verzeichnis)/Linkzähler (Datei)
  - Besitzer, Gruppenbesitzer
  - Speicherplatzverbrauch
  - Datum der letzten Modifikation
  - Name

Sonderrechte Unix (1)

SUID (set user ID):

- Erweitertes Zugriffsrecht für Dateien
- Unprivilegierte Benutzer erlangen kontrollierten Zugriff auf privilegierte Ressourcen
- Ausführung mit den Rechten des Besitzers der Datei (anstatt mit den Rechten des ausführenden Benutzers)
- Optische Notation bei ls:

  -rwsr-x---

Sonderrechte Unix (2)

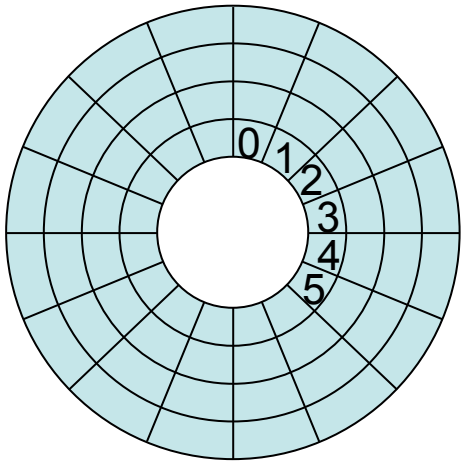
SGID (set group ID)

- Ausführung mit den Rechten der Gruppe, der die Datei/das Verzeichnis gehört (anstatt mit den Rechten der Gruppe, die ausführt)
- Verzeichnisse: Neu angelegte Dateien gehören der Gruppe, der auch das Verzeichnis gehört (anstatt der Gruppe, die eine Datei erzeugt)
- Optische Notation bei ls:

  drwxrws---

Implementierung von Dateisystem

- Festplatten sind eingeteilt in Blöcke, die durchnummeriert sind



Realisierung von Dateien

Drei verschiedene Alternativen zur Realisierung von Dateien:

- Zusammenhängende Belegung
- Belegung durch verkettete Listen
- I-Nodes

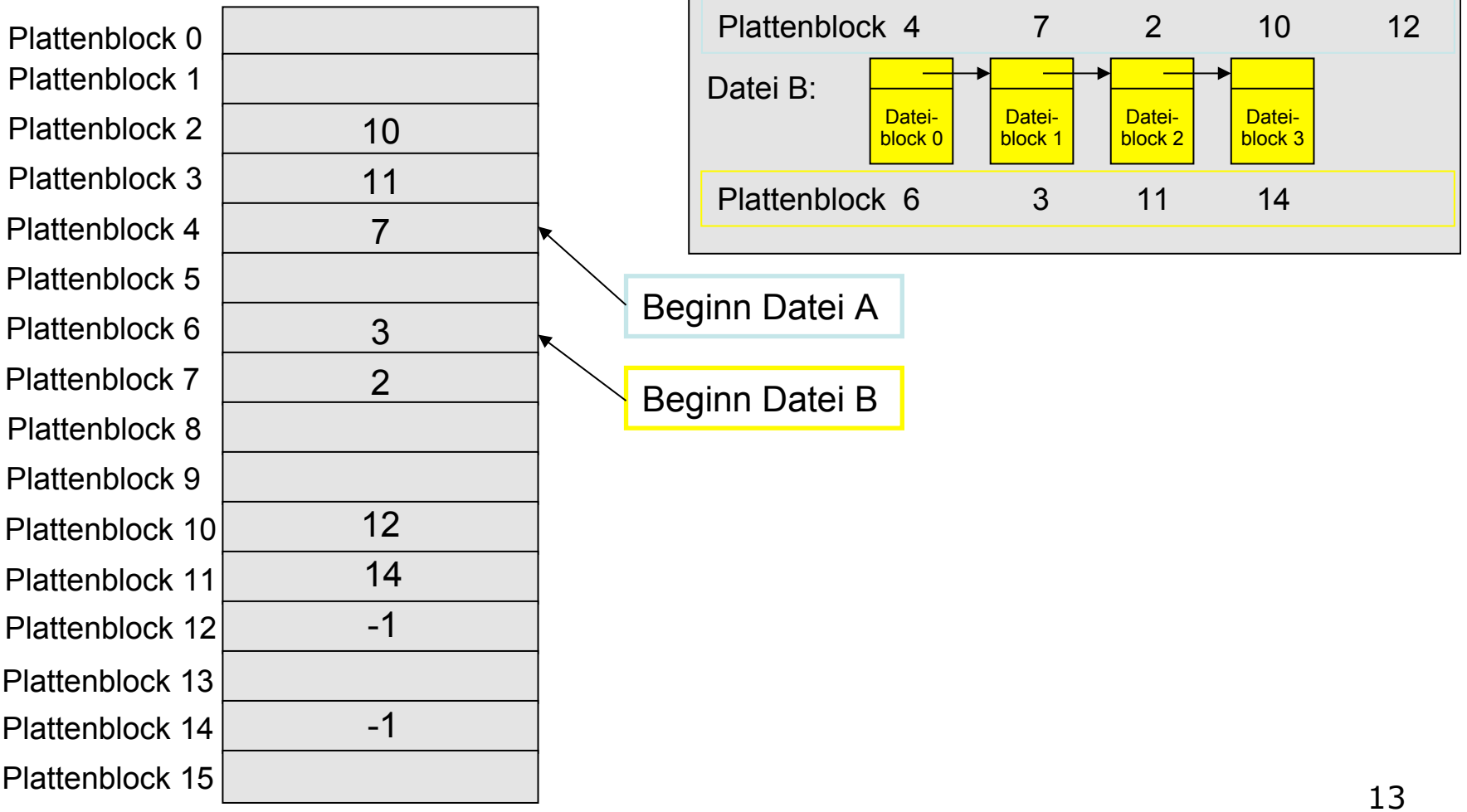
Verkettete Listen

- Fragmentierung der Festplatte führt nicht zu Verlust von Speicherplatz
- Dateien beliebiger Größe können angelegt werden (solange genug Plattenplatz vorhanden)
- Bei Zugriff auf Dateiblock n sind n-1 Lesezugriffe auf die Platte nötig, um den Block zu lokalisieren

FAT-System

- Datei-Allokationstabelle (FAT=File Allocation Table)
- Information über Verkettung der Blöcke im Hauptspeicher
- Bei wahlfreiem Zugriff auf Block n muss nur eine Kette von Verweisen im Hauptspeicher verfolgt werden (nicht auf der Platte)
- Nachteil: Anzahl der Einträge ist immer gleich der Gesamtzahl der Plattenblöcke
- z.B. FAT-16: 16 Bit zur Adressierung

FAT Beispiel



Beispiel: FAT-32

- 28 Bit zur Adressierung:
  2^{28} verschiedene Zeiger, je 4 Byte groß
- Größe der FAT: 2^{28} * 4 Byte = 2^{30}
  = 1 GB!

Fat-System

- Kleinere Blöcke führen zu weniger verschwendetem Platz pro Datei
- Je kleiner die Blockgröße, desto mehr Zeiger, desto größer die FAT im Hauptspeicher
- Maximale Größe des ganzen Dateisystems wird durch Blockgröße begrenzt
- FAT-16 muss z.B. für eine 2 GByte Partition eine Blockgröße von 32 KByte verwenden
- Andernfalls kann mit den 2^{16} verschiedenen Zeigern nicht die ganze Partition adressiert werden

Realisierung von Dateien
I-Nodes (1)

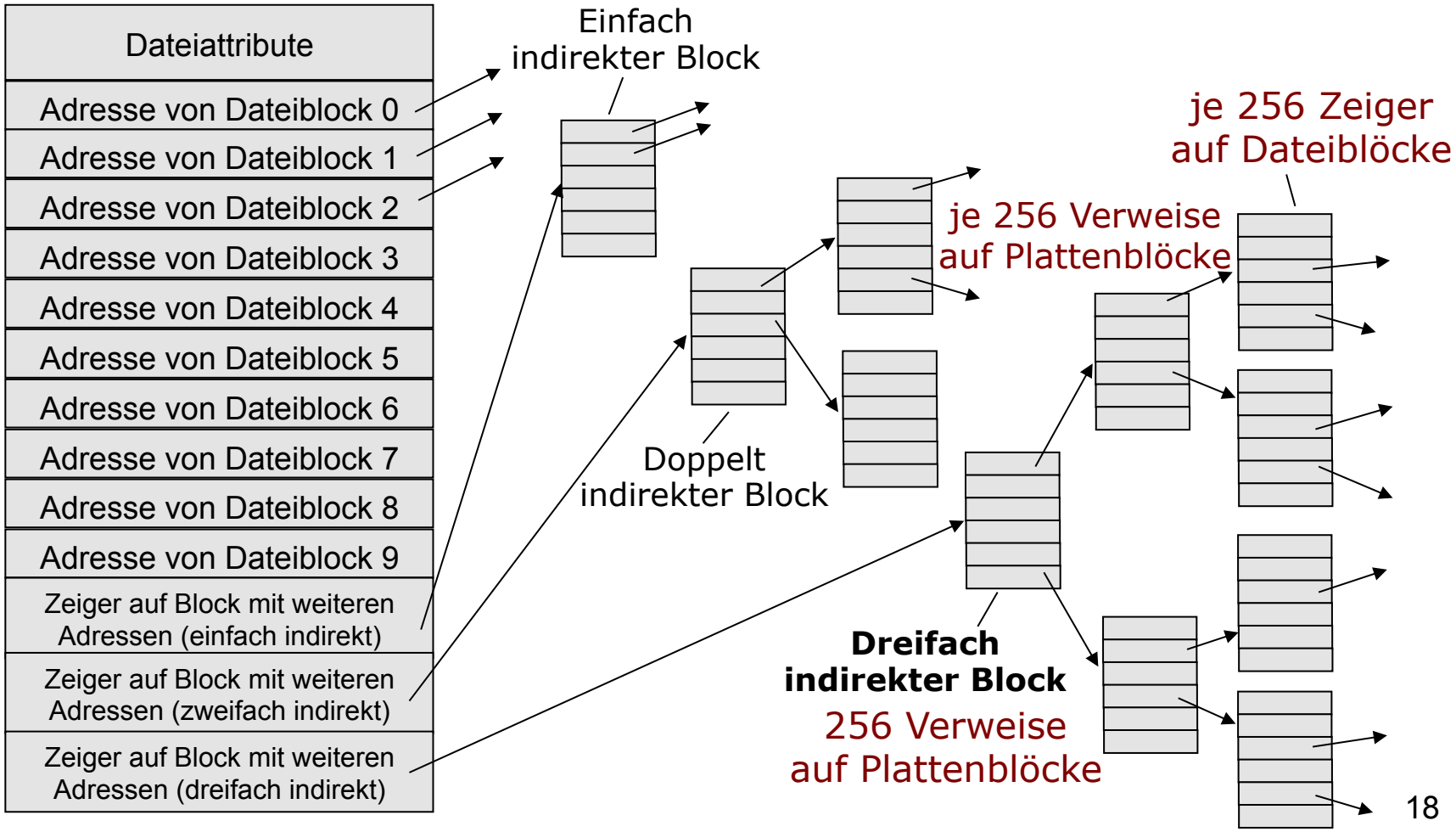
- Zu jeder Datei gehört eine Datenstruktur, der sogenannte I-Node (Index-Knoten, bei NTFS: File Record)
- I-Node enthält: Metadaten und Adressen von Plattenblöcken
- I-Node ermöglicht Zugriff auf alle Blöcke der Datei
- I-Node muss nur dann im Speicher sein, wenn eine Datei offen ist
- Wenn *k* Dateien offen und I-Node *n* Bytes benötigt: *k* * *n* Byte an Speicher benötigt

Realisierung von Dateien
I-Nodes (2)

I-Node einer Datei enthält

- Alle Attribute der Datei
- *m* Adressen von Plattenblöcken
  (UNIX System V: *m*=10, ext2/3: m=12)
- Verweise auf Plattenblöcke, die weitere Verweise auf Blöcke enthalten
- Auf die ersten *m* Plattenblöcke kann schnell zugegriffen werden
- Für die folgenden Plattenblöcke sind zusätzliche Zugriffe nötig

Realisierung von Dateien
I-Nodes (3)



Beispiel: Maximale Dateigröße

- Blockgröße 1 KByte, Zeigergröße 4 Bytes
- 10 direkte Zeiger des I-Nodes: 10 KByte Daten lassen sich speichern
- Einfach indirekter Zeiger verweist auf einen Plattenblock, der maximal 1 KByte/4Byte Zeiger verwalten kann, also 256 Zeiger
- Indirekt: 1 KByte * 256 = 256 KByte Daten
- Zweifach indirekter Zeiger:
  1 KByte * 256 * 256 = 64 MByte Daten
- Dreifache indirekter Zeiger:
  1 KByte * 256 * 256 * 256 = 16 GByte Daten

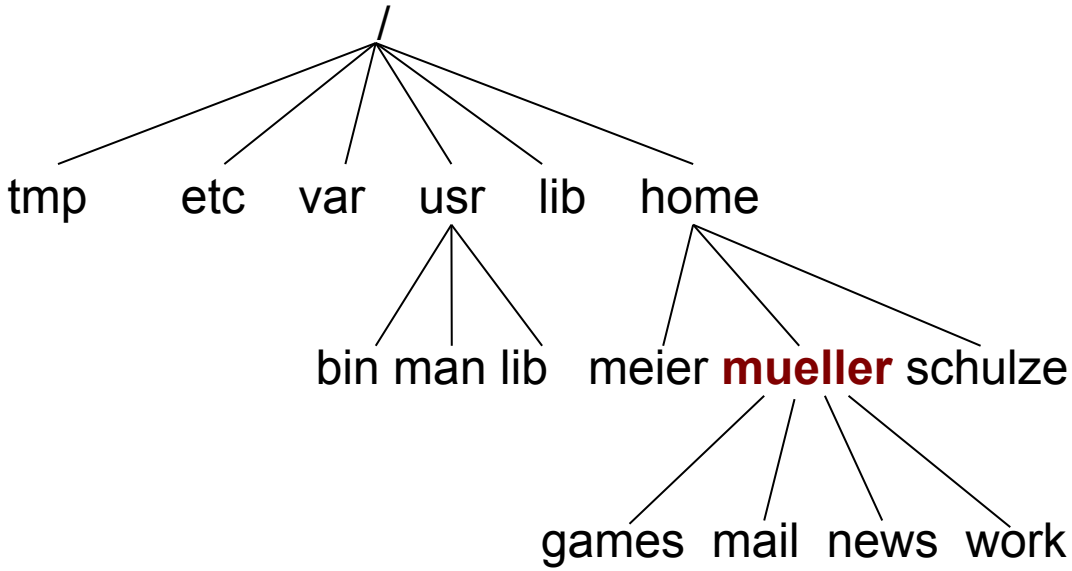
I-Nodes

- Im Betriebssystem existiert eine Tabelle mit allen Inodes
- Die Größe der I-Node-Tabelle wird beim Anlegen des Dateisystems festgelegt
- Wenn verfügbarer Plattenplatz oder alle I–Nodes belegt sind: Dateisystem voll
- Es muss eine ausreichende Anzahl von I-Nodes eingeplant werden

Realisierung von Verzeichnissen (1)

- Verzeichnisse sind ebenfalls Dateien
- Sie liefern eine Abbildung von Datei- bzw. Verzeichnisnamen auf I-Node-Nummern
- Jeder Verzeichniseintrag ist ein Paar aus Name und I-Node-Nummer
- Über die I-Nodes kommt man dann zu Dateiinhalten

Realisierung von Verzeichnissen (2)



aktuelles Vorgänger-verzeichnis

.	7
..	3
games	45
mail	76
news	9
work	14

I-Node von mueller

I-Node von home

I-Node Nr. 45

I-Node Nr. 76

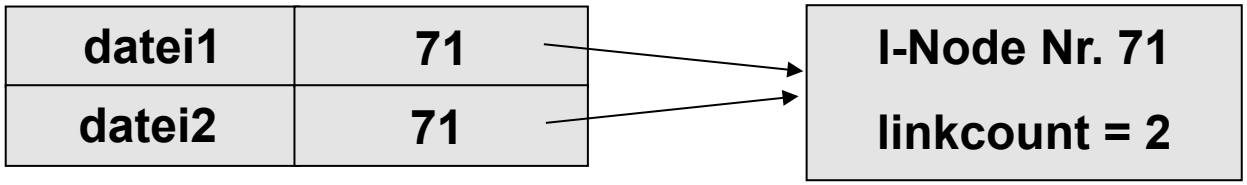
I-Node Nr. 9

I-Node Nr. 14

Zeiger auf die Datenblöcke der Datei

Implementierung von Hardlinks

```
$ ls –l
-rw-r----- 1 meier users … datei1
$ ln datei1 datei2
$ ls –l
-rw-r----- 2 meier users … datei1
-rw-r------ 2 meier users … datei2
```



Implementierung symbolischer Links

$ ls –l

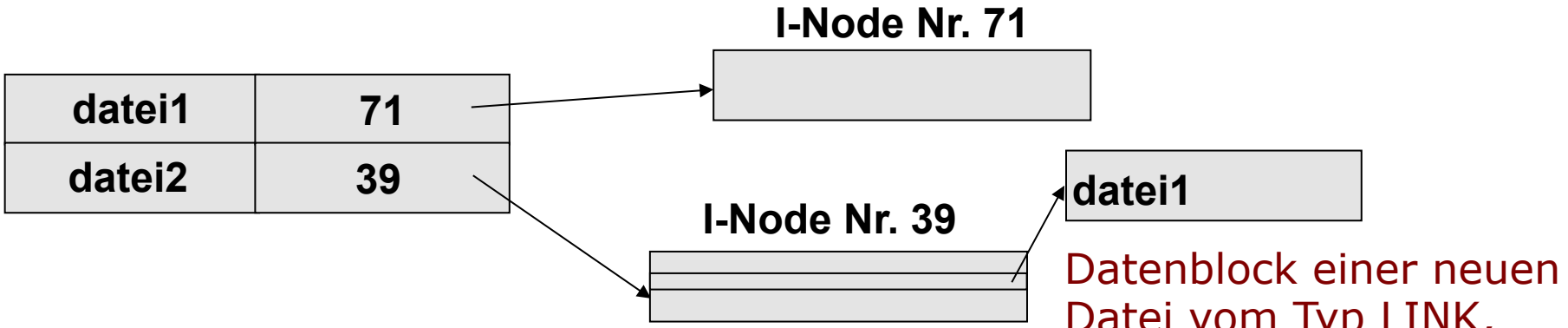
-rw-r----- **1** meier users … datei1

$ ln –s datei1 datei2

$ ls –l

-rw-r----- **1** meier users ... datei1

lrwxrwxrwx **1** meier users ... datei2 -> datei1



Prozesse

„Programm in Ausführung“

- Prozess = Instanz eines Programms mit
  - Aktuellem Wert vom Befehlszähler
  - Registerinhalten
  - Belegung von Variablen
- Multitasking-Betriebssysteme: Mehrere Prozesse können „pseudo-parallel“ (oder quasiparallel) ausgeführt werden

Motivation Multitasking

- Ein Benutzer will mehrere Aufgaben „gleichzeitig“ durchführen
- Mehrere Benutzer teilen sich einen leistungsfähigen Rechner: Timesharing-Systeme als spezielle Form des Multitasking
- Wartezeiten auf Ein-/Ausgaben kann durch Abarbeitung anderer Prozesse genutzt werden
- Ein-Programm-Lösung mit gleicher Funktionalität hätte häufig verworrene Kontrollstruktur

Prozesswechsel

- Nicht-präemptive Betriebssysteme: Prozessen kann nur dann der Prozessor entzogen werden, wenn sie ihn selbst abgeben
- Präemptive Betriebssysteme: Der aktive Prozess kann unterbrochen werden vom Betriebssystem
- Mehr Verwaltung, aber bessere Auslastung der CPU bei präemptiven Systemen

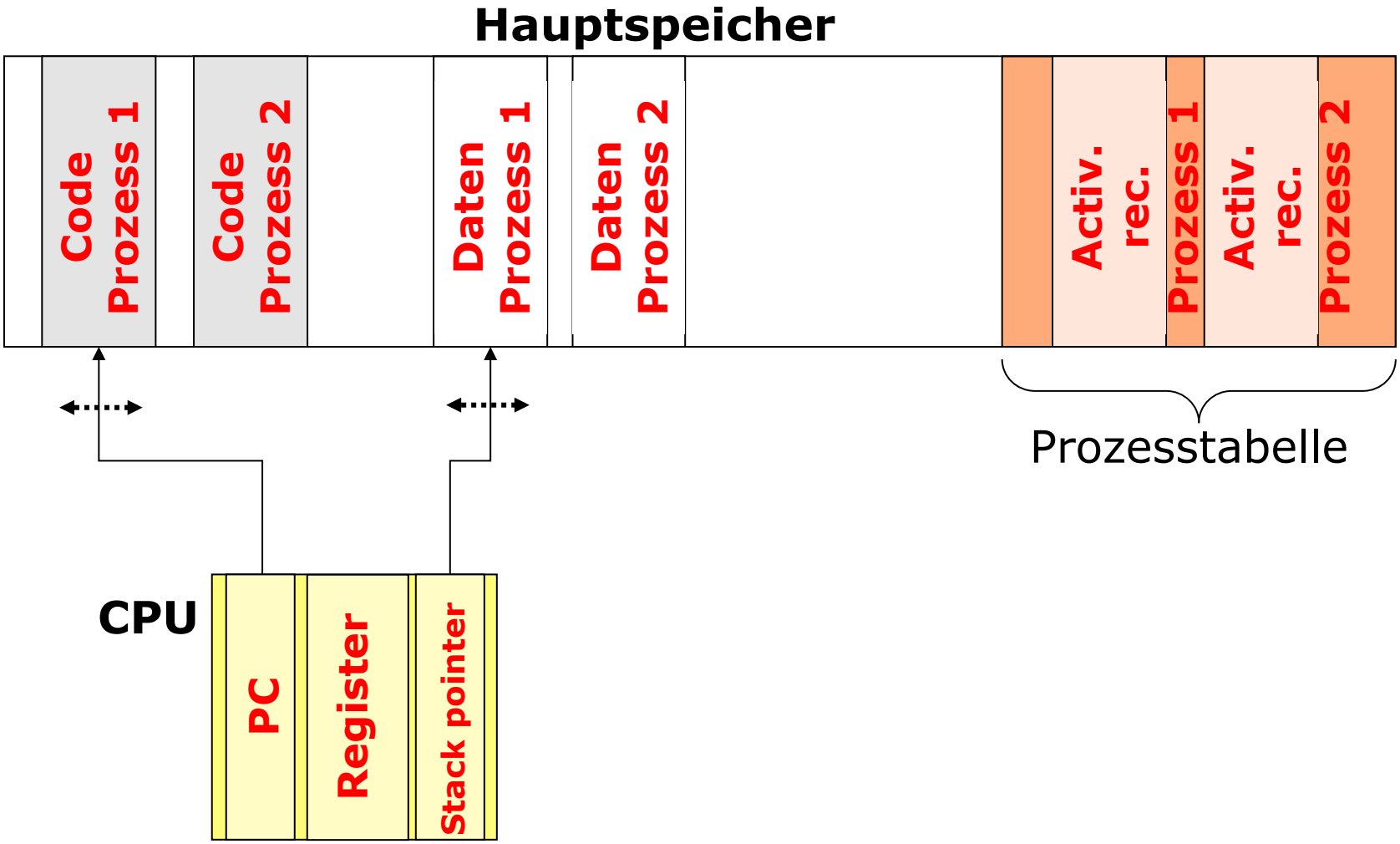
Adressraum

- Zu jedem Prozess gehört ein Adressraum im Hauptspeicher
- Liste von Speicherzellen mit Adressen, in denen der Prozess lesen und schreiben darf
- Adressraum enthält
  - Ausführbares Programm
  - Programmdaten
  - Kellerspeicher (“Stack”, für lokale Variablen)

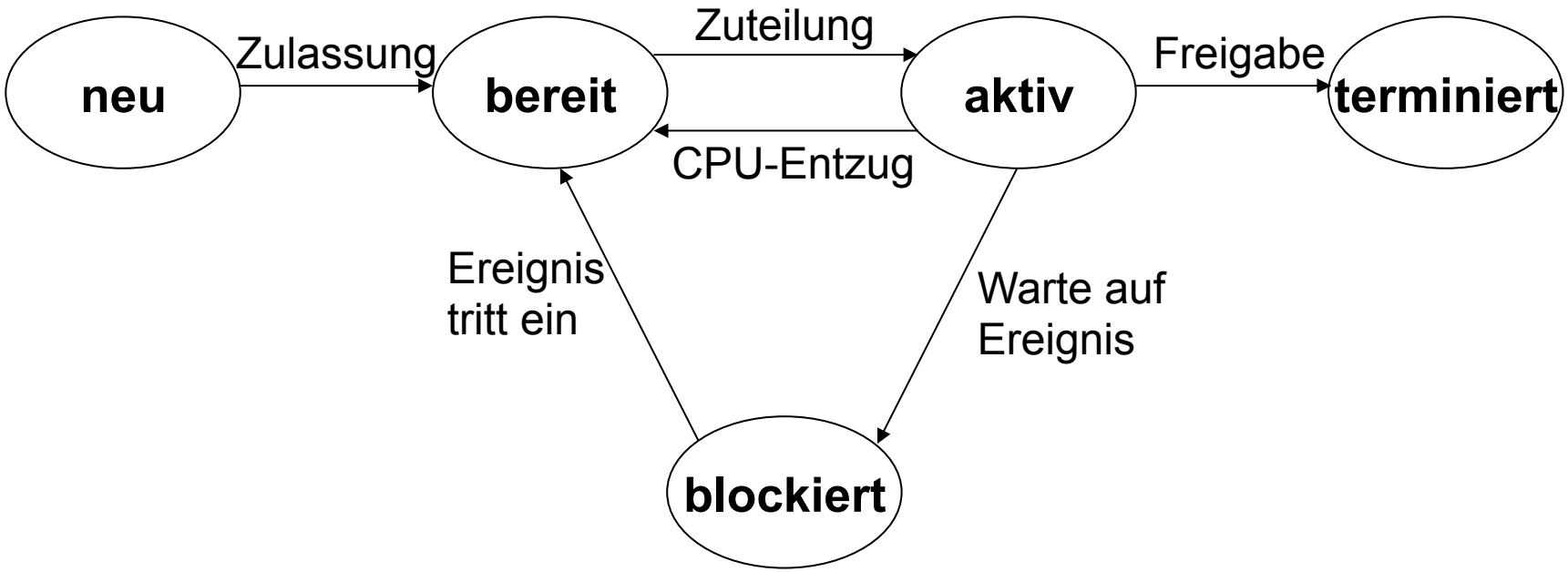
Prozessinformationen

- Individuelle Prozessinformationen von Prozessorregistern:
  - Befehlszähler
  - Allgemeine Register
  - Stack pointer (zeigt auf oberstes Kellerelement)
- Prozess ID, Priorität, Zustand, geöffnete Dateien, Startzeit, etc.
- Diese Informationen sind in einer sog. Prozesstabelle gespeichert („activation record”)

Beispiel: Prozesswechsel



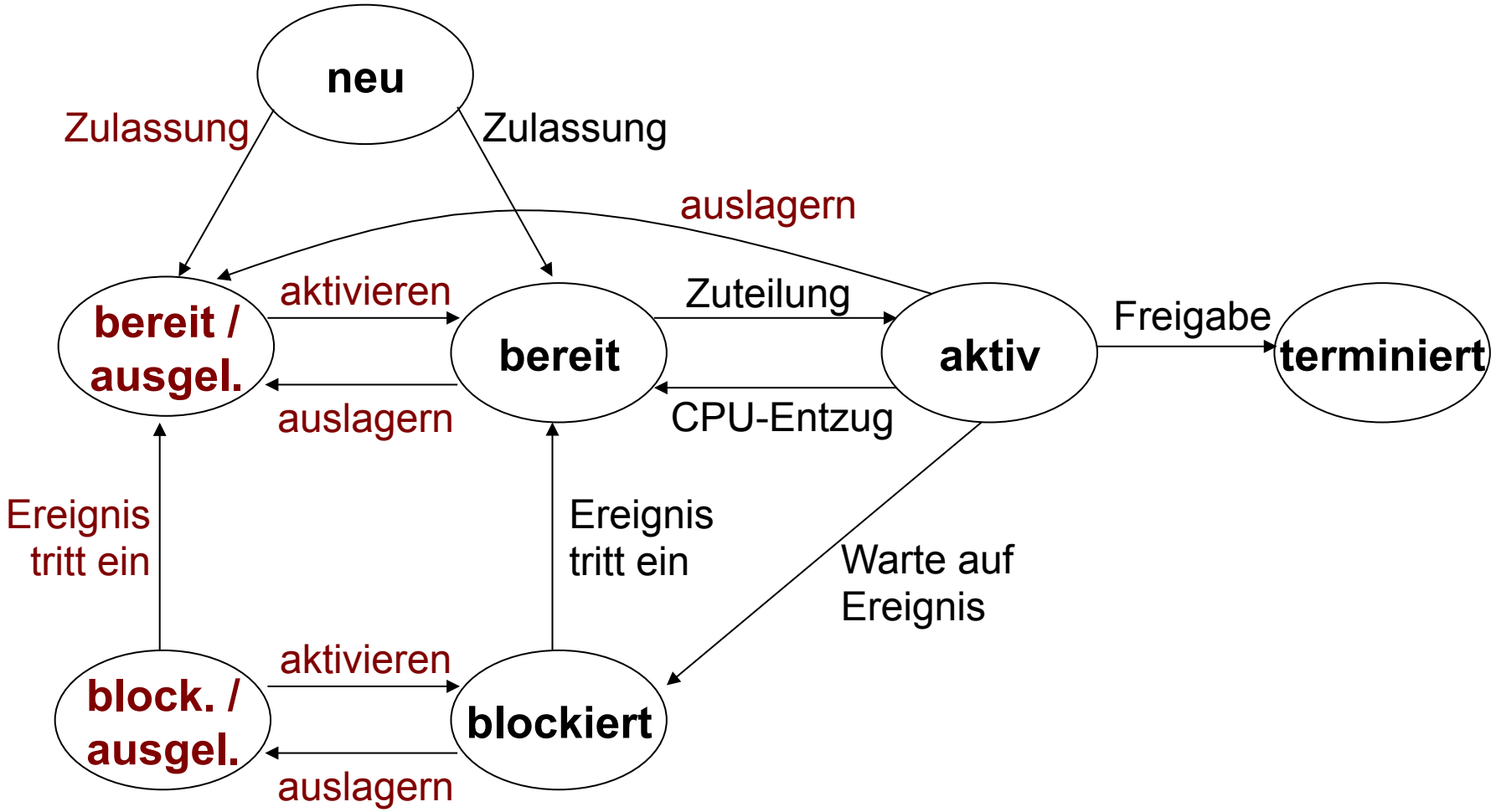
Prozessmodell mit 5 Zuständen



Swapping (Auslagern)

- Prozesse werden aus dem Hauptspeicher entfernt
- Daten von bereiten oder auf ein Ereignis wartenden Prozessen werden komplett auf die Festplatte ausgelagert
- Motivation:
  - Der Prozessor kann sich trotz Multitasking die meiste Zeit im Leerlauf befinden
  - Platz schaffen für die Aufnahme neuer Prozesse

Prozessmodell mit 7 Zuständen



Nebenläufigkeit und wechselseitiger Ausschluss

Anforderungen an wechselseitigen Ausschluss

- Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Regionen sein
- Es dürfen keine Annahmen über Geschwindigkeit und Anzahl der CPUs gemacht werden
- Kein Prozess, der außerhalb seiner kritischen Regionen läuft, darf andere Prozesse blockieren
- Kein Prozess sollte ewig darauf warten müssen, in seine kritische Region einzutreten

Lösungen für wechselseitigen Ausschluss

- Software-Lösungen: Verantwortlichkeit bei Prozessen, Anwendungsprogramme gezwungen sich zu koordinieren
- Hardware-Unterstützung: Spezielle Maschinenbefehle, reduzieren Verwaltungsaufwand
- Ins Betriebssystem integrierte Lösungen

Wechselseitiger Ausschluss in Software

- Wechselseitiger Ausschluss ist in Software schwer zu realisieren
- Alles was einfacher ist als Petersons Algorithmus ist höchstwahrscheinlich falsch
- Formale Beweise sind unabdingbar!
- Software-Lösungen für wechselseitigen Ausschluss benötigen aktives Warten

Wechselseitiger Ausschluss in Hardware (1)

- Neues Konzept:
  - Einführung atomarer Operationen
  - Hardware garantiert atomare Ausführung
- Testen und Setzen zusammen bilden eine atomare Operation:
  - Befehl TSL (Test and Set Lock)
  - Da TSL ein einziger Befehl ist, kann ein Prozesswechsel nicht zwischen Testen und Setzen erfolgen

Wechselseitiger Ausschluss in Hardware (2)

- Vorteil: Wechselseitiger Ausschluss ist garantiert
- Nachteil: Aktives Warten wie bei Software-Lösungen

Wechselseitiger Ausschluss, ins Betriebssystem integriert

- Prozesse blockieren statt warten
- Systemaufrufe statt Verschwendung von Rechenzeit
- Systemaufruf sleep(id) zum Blockieren von Prozessen (id: eindeutige Bezeichnung für kritische Region)
- Systemaufruf wakeup(id) nach Verlassen des kritischen Abschnitts:

Mutexe (1)

- Ein Mutex m besteht aus einer binären Lock-Variable $lock_m$ und einer Warteschlange $queue_m$
- Vor Eintritt in die kritische Region wird die Funktion mutex_lock(m) aufgerufen
- Darin: Überprüfung, ob die kritische Region schon belegt ist
  - Falls ja: Prozess blockiert (sleep) und wird in Warteschlange eingefügt
  - Falls nein: Lock-Variable wird gesetzt und Prozess darf in kritische Region eintreten

Mutexe (2)

- Einfachste Möglichkeit um wechselseitigen Ausschluss mit Systemfunktionen zu garantieren
- Zwei Zustände: gesperrt / nicht gesperrt
- Kein aktives Warten, CPU wird abgegeben
- Der Prozess, der den Mutex gesperrt hat, gibt ihn auch wieder frei

Semaphore

- Wert des Semaphors repräsentiert die Anzahl der Weckrufe, die ausstehen
- Drei mögliche Situationen für den Wert eines Semaphors:
  - Wert < 0: weitere Weckrufe stehen schon aus, nächster Prozess legt sich auch schlafen
  - Wert 0: keine Weckrufe sind bisher gespeichert, nächster Prozess legt sich schlafen
  - Wert > 0: frei, nächster Prozess darf fortfahren

Semaphor: down/up

- down-Operation:
  - Verringere den Wert von $count_s$ um 1
  - Wenn $count_s$<0, blockiere den aufrufenden Prozess (Warteschlange), sonst fahre fort
- up-Operation:
  - Erhöhe den Wert von $count_s$ um 1
  - Wenn $count_s$<=0, wecke einen der blockierten Prozesse auf
  - Bedeutung: Wenn $count_s$<0 vor Erhöhen: Es gibt mindestens einen wartenden Prozess

Deadlocks

Deadlocks

- Ressourcen werden nach und nach von den Prozessen angefordert (und nach Benutzung wieder freigegeben)
- Dabei kann es dazu kommen, dass eine Menge von Prozessen sich in einem Deadlock befindet

Voraussetzungen für Ressourcen-Deadlocks

- Wechselseitiger Ausschluss:
  Jede Ressource ist entweder verfügbar oder genau einem Prozess zugeordnet

Voraussetzungen für Ressourcen-Deadlocks

- Wechselseitiger Ausschluss
- Besitzen und Warten:
  Prozesse, die schon Ressourcen reserviert haben, können noch weitere Ressourcen anfordern

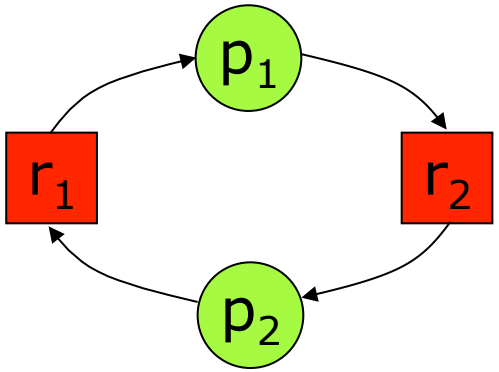
Voraussetzungen für Ressourcen-Deadlocks

- Wechselseitiger Ausschluss
- Besitzen und Warten
- Kein Ressourcenentzug:
  Ressourcen, die einem Prozess bewilligt wurden, können nicht gewaltsam wieder entzogen werden

Voraussetzungen für Ressourcen-Deadlocks

- Wechselseitiger Ausschluss
- Besitzen und Warten
- Kein Ressourcenentzug
- Zyklisches Warten:
  Es gibt eine zyklische Kette von Prozessen, von denen jeder auf eine Ressource wartet, die dem nächsten Prozess in der Kette gehört

Belegungs-Anforderungs-Graph

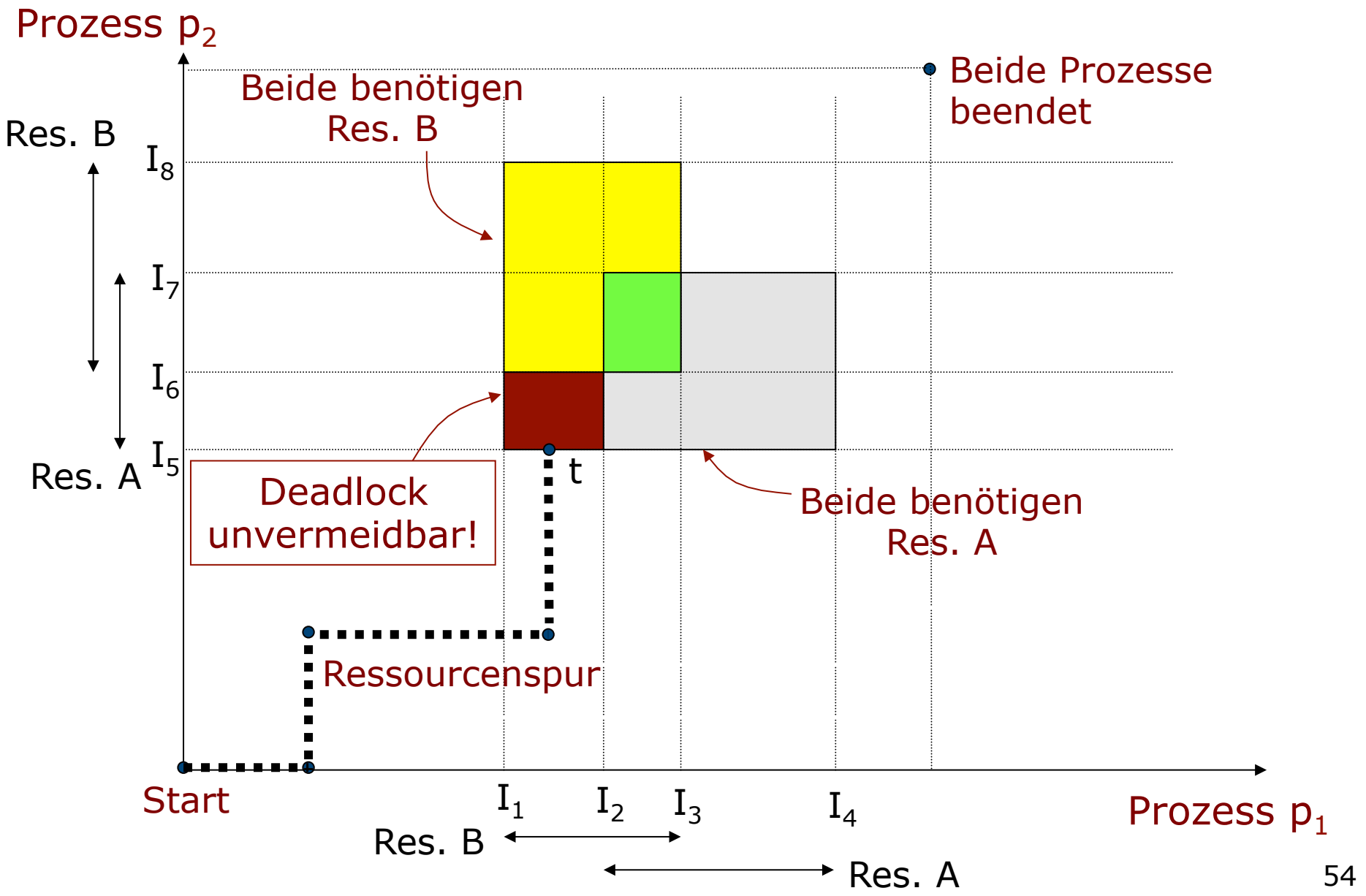


- Zyklen im Belegungs-Anforderungsgraphen repräsentieren Deadlocks!

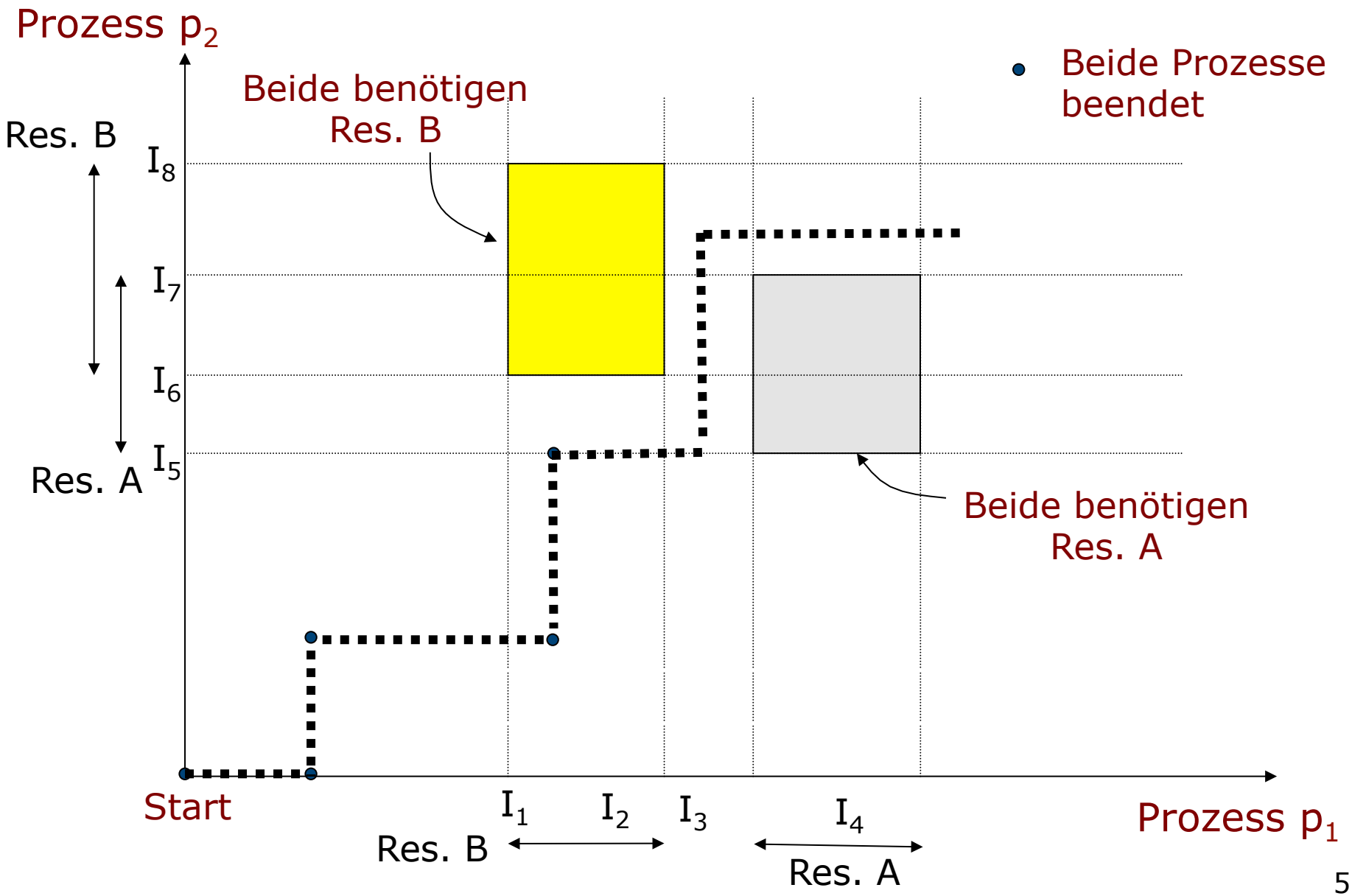
Ressourcendiagramm

- Diagramm zur Visualisierung der Ressourcenanforderungen über die Zeit
- Besitzt Zeitachsen für die Prozesse
- Dient zur Erkennung von potentiellen Deadlocks
- Ressourcenspur: Eine mögliche Ausführungsreihenfolge der Anweisungen

Ressourcendiagramm: Deadlock



Beispiel: Nie Deadlock (2)



Verhindern von Deadlocks

- Bankier-Algorithmus: Ressourcenaufteilung an Prozesse, so dass Deadlocks garantiert verhindert werden können
- Voraussetzungen:
  - Es ist im Voraus bekannt, welche und wie viele Ressourcen die einzelnen Prozesse (maximal) anfordern werden
  - Diese maximale Anforderung übersteigt für keinen Prozess die zur Verfügung stehenden Ressourcen

Bankier-Algorithmus (1)

Ein Zustand ist sicher, wenn

- Es auf jeden Fall eine deadlockfreie „Restausführung“ aller Prozesse gibt
- Unabhängig davon, in welcher Weise die Prozesse in Zukunft ihre Ressourcen-anforderungen und -freigaben durchführen
- Auch dann wenn gilt
  - Prozesse stellen ihre restlichen Anforderungen jeweils auf einen Schlag
  - Freigaben gibt es erst bei Prozessbeendigung (Worst Case)

Bankier-Algorithmus (2)

- Wenn deadlockfreie Restausführung nicht garantiert werden kann: Zustand ist unsicher
- Beachte: Ein unsicherer Zustand muss nicht notwendigerweise zu einem Deadlock führen! (Wir betrachten den Worst Case)
- Bei sicheren Zuständen kann garantiert werden, dass es eine Reihenfolge gibt, s.d. alle Prozesse zu Ende laufen können!

Bankier-Algorithmus (3)

Probleme bei der praktischen Anwendung:

- Prozesse können meist nicht im Voraus eine verlässliche Obergrenze für ihre Ressourcenanforderungen geben
- Anzahl der Prozesse ist nicht fest, sondern ändert sich ständig (z.B. durch Ein- / Ausloggen)

Scheduling

Kurzfristiges Scheduling

- Der kurzfristige Scheduler weist die CPU verschiedenen (konkurrierenden) Prozessen zu, um “optimale Performance” zu erzielen
- Mehrere verschiedene Optimierungsziele sind denkbar, verschiedene Scheduling-Algorithmen existieren

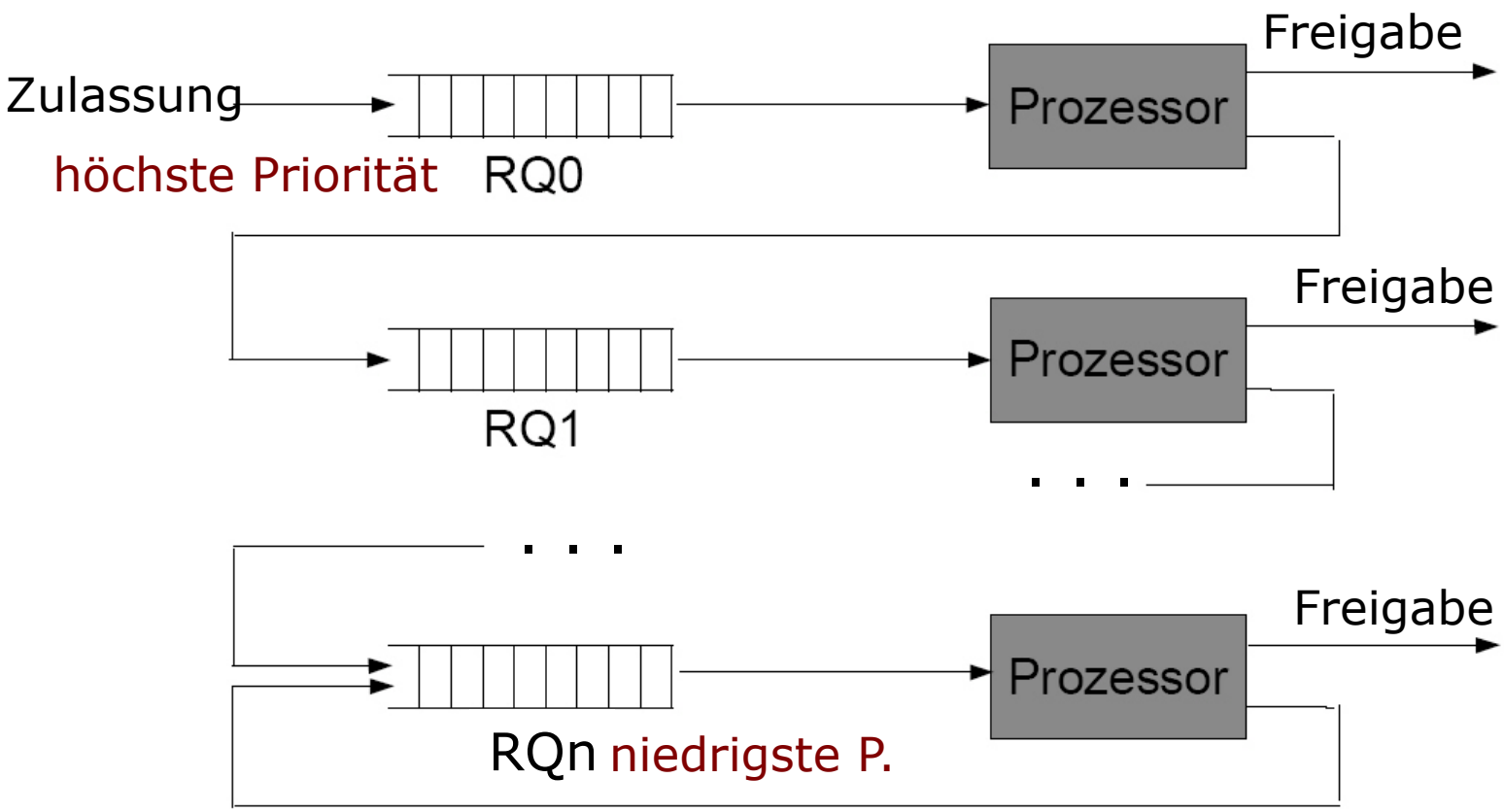
Kriterien für das kurzfristige Scheduling

- Benutzerorientiert:
  - Minimale Antwortzeit bei interaktivem System
  - Minimale Zeit zwischen Eingang und Abschluss eines Prozesses (Durchlaufzeit)
  - Gute Vorhersagbarkeit (unabhängig von Systemauslastung ähnliche Zeit)
- Systemorientiert:
  - Maximale CPU-Auslastung (aktive Zeit)
  - Maximale Anzahl von Prozessen, die pro Zeiteinheit abgearbeitet werden (Durchsatz, pro Stunde)

Feedback Queues (1)

- Präemptiv (Zeitintervall), dynamische Prioritäten
- Wenn ein Prozess die CPU abgeben muss, dann wird er in eine Warteschlange mit der nächst geringeren Priorität eingefügt
- Dadurch: Verbrauchte CPU-Zeit wird angenähert durch Anzahl erzwungener CPU-Abgaben
- Abarbeitung der Warteschlangen nach Priorität

Feedback Queues (2)



Scheduling bei UNIX

- Es gibt verschiedene Warteschlangen mit unterschiedlichen Prioritäten
- Ausgeführt wird anfangs der erste Prozess in der nichtleeren Warteschlange mit höchster Priorität
- Die Prozesse höchster Priorität werden anschließend untereinander nach Round Robin gescheduled (z.B. jede 100ms)
- Neuberechnung der Prioritäten in regelmäßigen Zeitabständen (z.B. jede Sekunde), Wartezeit geht in die Priorität ein

Klausur

- Datum: 14. März 2013, 9:00 s.t.
- Bitte UniCard mitbringen
- Räume: Gebäude 101, Räume 00-026, 00-036 und 00-010/14
- Klausureinsicht: 27. März, 13.30-15.30 Uhr in Gebäude 74, Raum 00-009 (Küche)
- Viel Glück!