

# Betriebssysteme

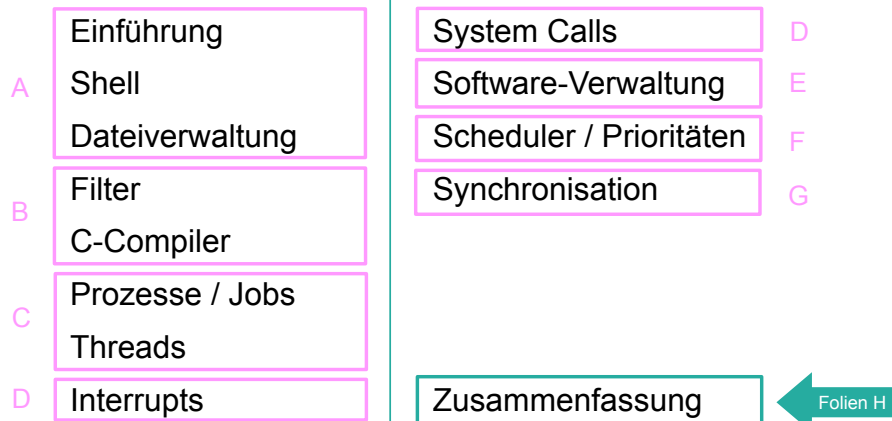
WS 2014/15

**Hans-Georg Eßer**  
Dipl.-Math., Dipl.-Inform.

**Foliensatz H:**  
• Zusammenfassung

v1.0, 2015/01/10

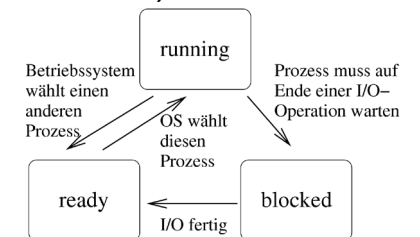
## Übersicht: BS Praxis und BS Theorie



# Zusammenfassung

## Prozesse

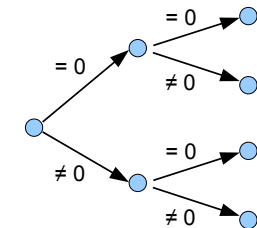
- Abstraktion: Programm, das ausgeführt wird
- separater Speicher (Adressraum)
- Prozesskontrollblock
- Standard-Zustände:
- Hierarchie (Vater / Sohn)
- Linux: Vordergrund, Hintergrund, Job vs. Prozess, nohup, disown
- Prioritäten; Linux: nice, renice



- „Aktivitätsstrang“ in einem Prozess
- Threads eines Prozesses teilen den Speicher
- User Level vs. Kernel Level Threads
  - UL: Verwaltung komplett im User Mode; billig; schneller Kontextwechsel
  - KL: Verwaltung im Kernel; teurer; aber: andere Threads bleiben aktiv, wenn einer auf I/O blockiert
- Thread-Zustände (nicht alle Prozess-Zustände)

- **Prozesse:**
  - fork → echtes Verdoppeln (außer Details)
  - exec → ersetzt laufendes Programm im Prozess
  - wait → wartet auf Kind-Prozess
  - exit → Prozessende
- **Threads:**
  - pthread\_create (mit Funktion als Argument)
  - pthread\_join

```
int pid1 = fork();
printf ("%s\n", "[1] Ein Fork ist durch, einer muss noch.");
int pid2 = fork();
printf ("%s\n", "[2] Zeit für eine Fallunterscheidung.");
if ( (pid1==0) && (pid2==0) ) {
    printf ("%s\n", "[3] Ich starte jetzt emacs.");
    execl ("/bin/emacs", "/etc/fstab", (char *)NULL);
    int pid3 = fork();
    printf ("%s\n", "[4] Nach dem dritten Fork.");
} else {
    printf ("%s\n", "[5] Ich gucke nur zu.");
};
printf ("%s\n", "[6] Ende.");
```



- **Polling vs. Interrupts**
- I/O = asynchroner Interrupt
- Software Interrupts: Exceptions, System Calls
- Interrupt Handler (auch: Mehrfach-Interrupts)
- CPU-lastig vs. I/O-lastig
- Linux: top half + bottom half (Tasklet)

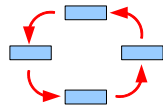
- Programmen Zugriff auf Kernel-Funktionen geben – aber kontrolliert
- System Call ist Interface in den Kernel
- realisiert über Software Interrupt (z. B. `int 0x80`)
- Weg von der Bibliotheksfunktion `fread` bis in den Kernel und zurück
- Standard-Syscalls (`open`, `read`, `write`, `close`, `fork`, `exec` etc.)

- **kooperativ** (nicht-unterbrechend) vs. **präemptiv** (unterbrechend)
- Batch: FCFS, SJF, SRT
- FCFS bevorzugt CPU-lastige Prozesse
- Burst-Dauer-Prognose (Mittelwert, exponent.)
- Interaktiv: RR, VRR, Prioritäten, Lotterie
- Auch RR bevorzugt CPU-lastige Proz. → VRR
- RR: **Quantum** geeignet wählen

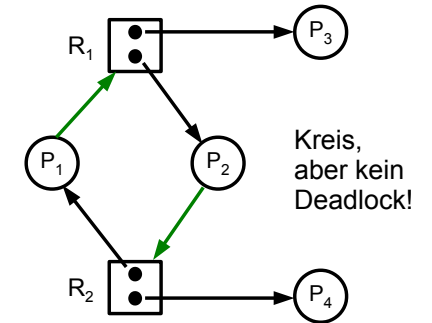
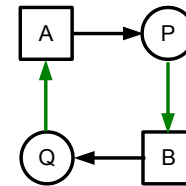
- **Kritischer Abschnitt:** Programmteil, der auf gemeinsame Daten zugreift
- **Gegenseitiger Ausschluss:** keine parallele Ausführung kritischer Abschnitte
- **TSL:** Test and Set Lock (CPU-Instruktion), arbeitet atomar
- **Aktives Warten** (Schleife) vs. **Passives Warten** („sleep & wake“)
- Erzeuger-Verbraucher-Problem

- **Semaphor:** Zählvariable
  - Ressource anfordern: `wait()`
  - Ressource wieder freigeben: `signal()`
  - Warteschlange für Prozesse, die nicht direkt eine Ress. erhalten
- **Mutex:** Mutual Exclusion
  - Semaphor mit Initialwert 1
  - → kritische Abschnitte
  - `wait()` → `lock()`, `signal()` → `unlock()`
  - auch hier Warteschlange

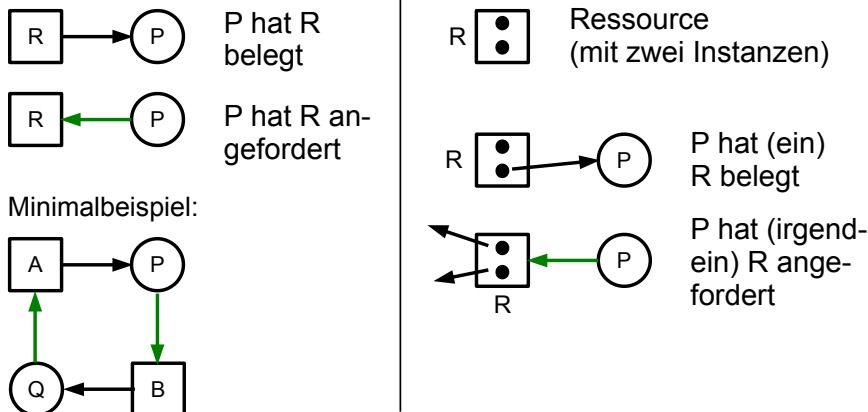
- Minimal-Beispiel: P: lock(A); lock(B)  
Q: lock(B); lock(A)
- Fünf-Philosophen-Problem
- unterbrechbare / nicht unterbr. Ressourcen**
- Deadlock genau dann wenn (1) – (4):  
(1) Gegens. Ausschluss (2) Hold and Wait  
(3) Ununterbrechbarkeit der Ressourcen  
(4) Zyklisches Warten



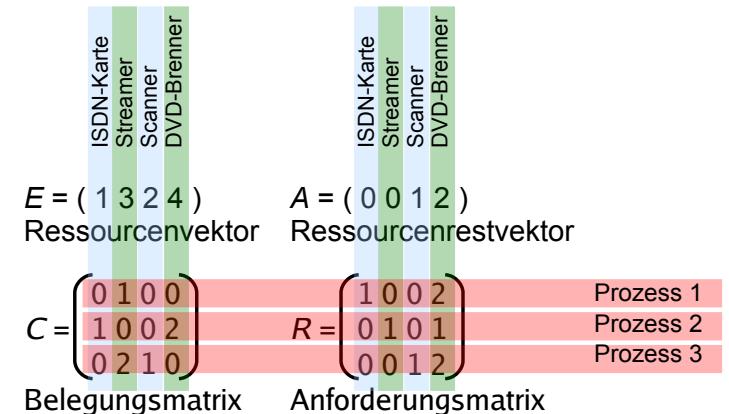
- Einfache Ressourcen:**  
Kreis im Graph  
 $\Leftrightarrow$  Deadlock
- Mehrfache Ressourcen:**  
Kreis im Graph  
 $\Leftarrow$  Deadlock (nicht  $\Rightarrow$ )



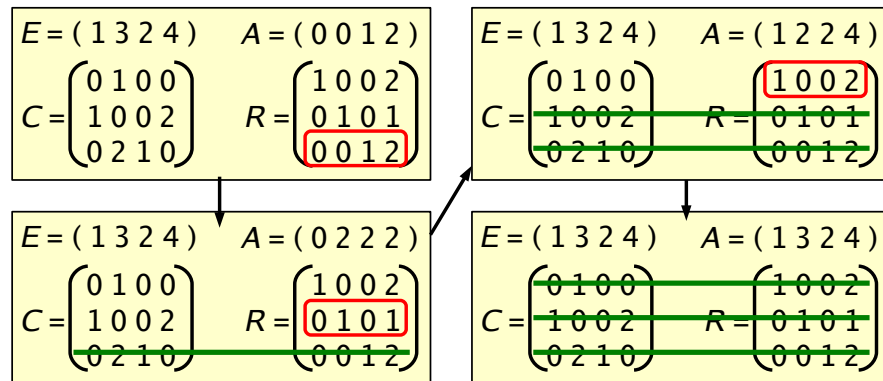
**Ressourcen-Zuordnungs-Graph:**



**Deadlock-Erkennung:**



- Alle Prozesse, die nach diesem Algorithmus nicht markiert sind, sind an einem Deadlock beteiligt
- Beispiel



- Deadlock-Vermeidung:
  - sichere vs. unsichere Zustände:
    - „Es gibt Ausführreihenfolge, die keinen Deadlock verursacht, wenn alle Prozesse sofort ihre maximalen Ressourcenforderungen stellen“
  - aus „unsicher“ folgt nicht zwingend Deadlock
  - Banker-Algorithmus: prüfen, ob Anforderung zu sicherem Zustand führt

- Deadlock-Verhinderung:
  - eine der vier Bedingungen verhindern
  - erfolgreich vor allem beim zyklischen Warten:
    - Ressourcen sortieren
    - Locks immer in gleicher Reihenfolge anfordern
    - (Beweis durch Widerspruch)

- Debian- und RPM-Pakete
- paket-basierte Tools (dpkg, rpm) vs.
  - installieren, aktualisieren, entfernen, Informationen
  - Abhängigkeiten und Konflikte
- repository-basierte Tools (apt-get, ...)
  - Repo hinzufügen (Drittanbieter?), löschen
  - Paket installieren, aktualisieren, entfernen etc.
  - Abhängigkeiten automatisch auflösen
  - Distributions-Upgrade (!: Update vs. Upgrade)

- Prompt (>, \$, #) – wann root?
- ~ = Home-Verzeichnis
- pwd, ls, touch, cd, cp, mv, rm, Wildcards (?, \*)
- absolute und relative Pfade
- mkdir, rmdir, rm -r
- kein „Undelete“
- man, vi
- set, export, echo (Shell-Variablen)

- history
- Filter: prog1 | prog2
- Umleitung: prog < eingabe > ausgabe
- cat, cut, fmt, split, sort, uniq, grep, sed
- reguläre Ausdrücke
- C-Grundlagen, gcc, Programm- und Header-Dateien (Implementation und Prototyp)