

# Betriebssysteme

WS 2015/16

**Hans-Georg Eßer**

**Foliensatz H:**

- Zusammenfassung

v1.3, 2015/12/18

# Übersicht: BS Praxis und BS Theorie

A	Einführung	System Calls	D
	Shell	<del>Software-Verwaltung</del>	E
	Dateiverwaltung	Scheduler / Prioritäten	F
B	Filter	Synchronisation	G
	C-Compiler	Speicherverwaltung	S
C	Prozesse / Jobs	<del>Dateisysteme, Zugriffsrechte</del>	T
	Threads		
D	Interrupts	Zusammenfassung	

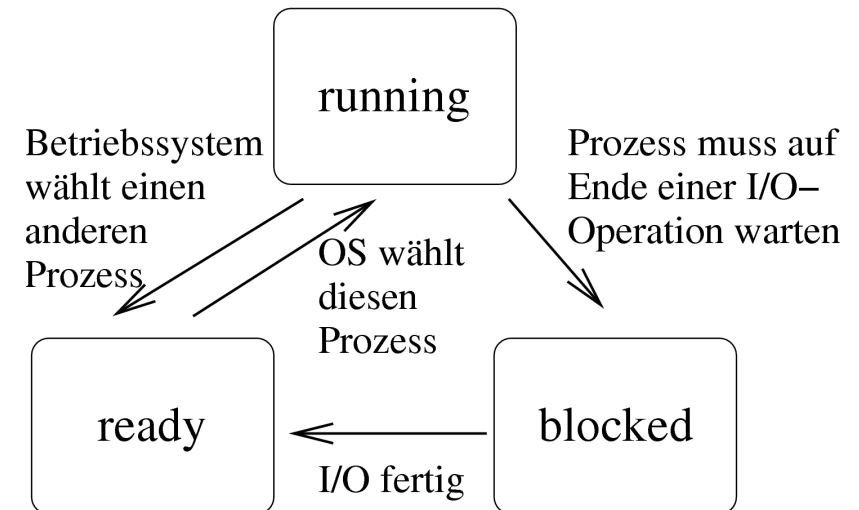


# Zusammenfassung

- 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)

- 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:**

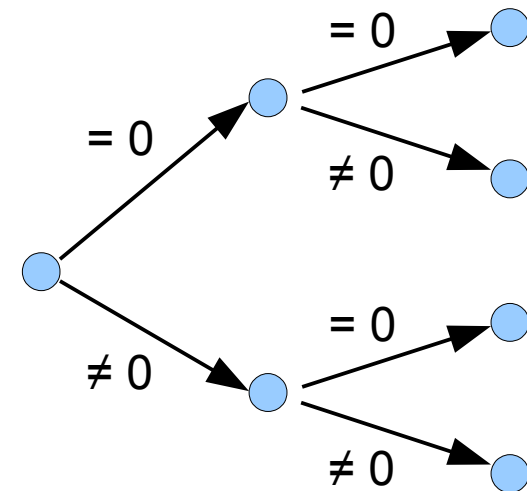
- `pthread_create` (mit Funktion als Argument)
- `pthread_join`



```

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)

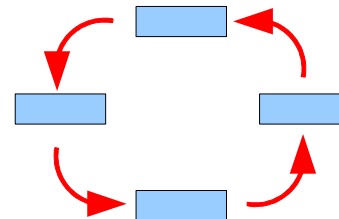
- Programmen Zugriff auf Kernel-Funktionen geben – aber kontrolliert
- System Call ist Interface in den Kernel
- realisiert über Software Interrupt (z. B. `int 0x80`)
- Weg von der Bibliotheksfunktion `fread` bis in den Kernel und zurück
- Standard-Syscalls (`open`, `read`, `write`, `close`, `fork`,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

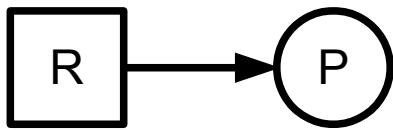
- **Kritischer Abschnitt:** Programmteil, der auf gemeinsame Daten zugreift
- **Gegenseitiger Ausschluss:** keine parallele Ausführung kritischer Abschnitte
- **TSL:** Test and Set Lock (CPU-Instruktion), arbeitet atomar
- **Aktives Warten** (Schleife) vs. **Passives Warten** („sleep & wake“)
- Erzeuger-Verbraucher-Problem

- **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

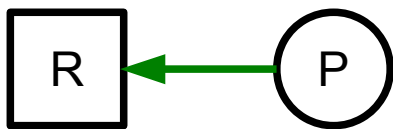
- Minimal-Beispiel: P: lock(A); lock(B)  
Q: lock(B); lock(A)
- Fünf-Philosophen-Problem
- **unterbrechbare / nicht unterbr. Ressourcen**
- Deadlock genau dann wenn (1) – (4):
  - (1) Gegens. Ausschluss
  - (2) Hold and Wait
  - (3) Ununterbrechbarkeit der Ressourcen
  - (4) Zyklisches Warten



- Ressourcen-Zuordnungs-Graph:**

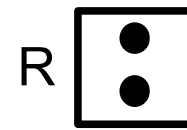
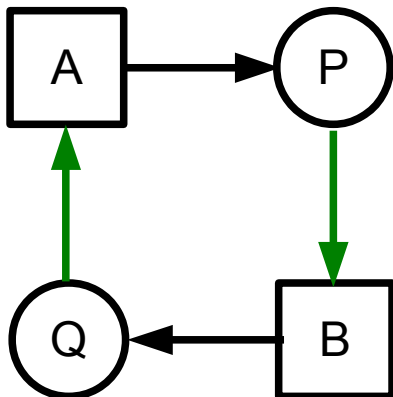


P hat R belegt

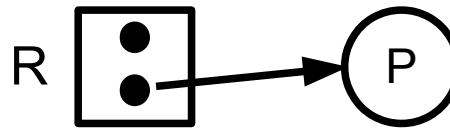


P hat R angefordert

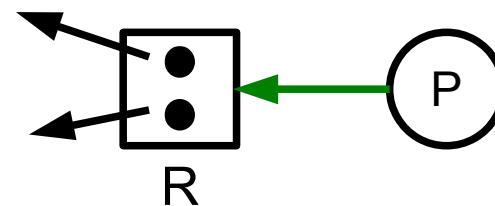
Minimalbeispiel:



Ressource (mit zwei Instanzen)



P hat (ein) R belegt



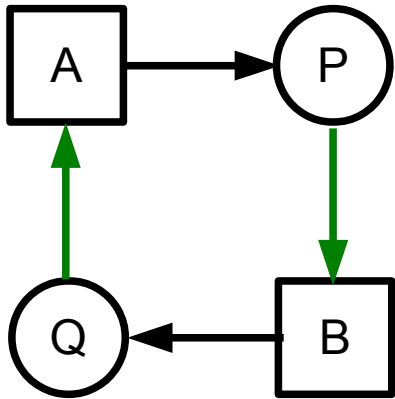
P hat (irgend-ein) R angefordert



# Deadlocks (3)

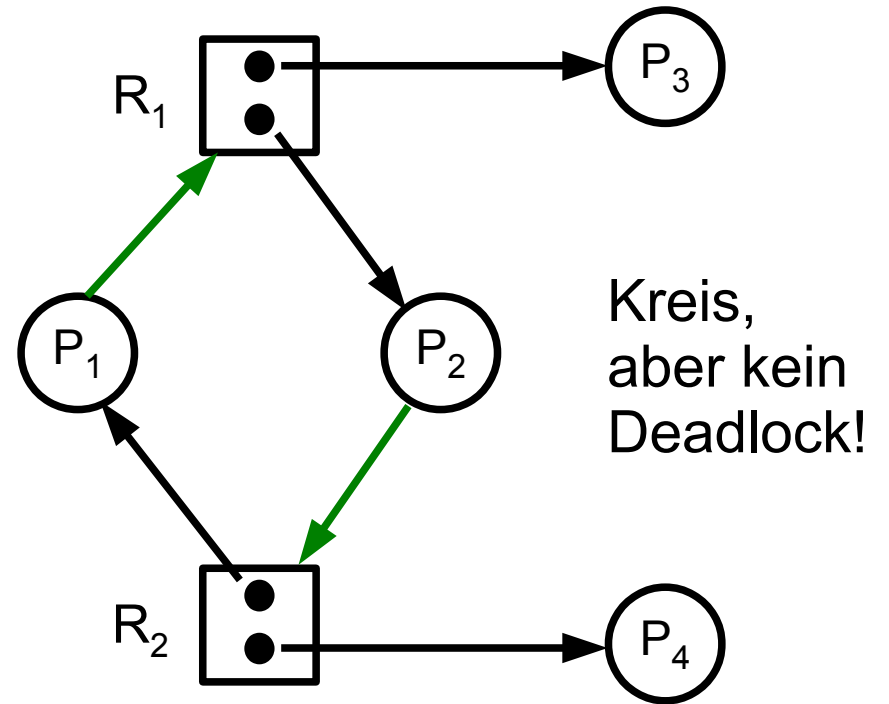
- **Einfache Ressourcen:**

Kreis im Graph  
 $\Leftrightarrow$  Deadlock



- **Mehrfache Ressourcen:**

Kreis im Graph  
 $\Leftarrow$  Deadlock (nicht  $\Rightarrow$ )



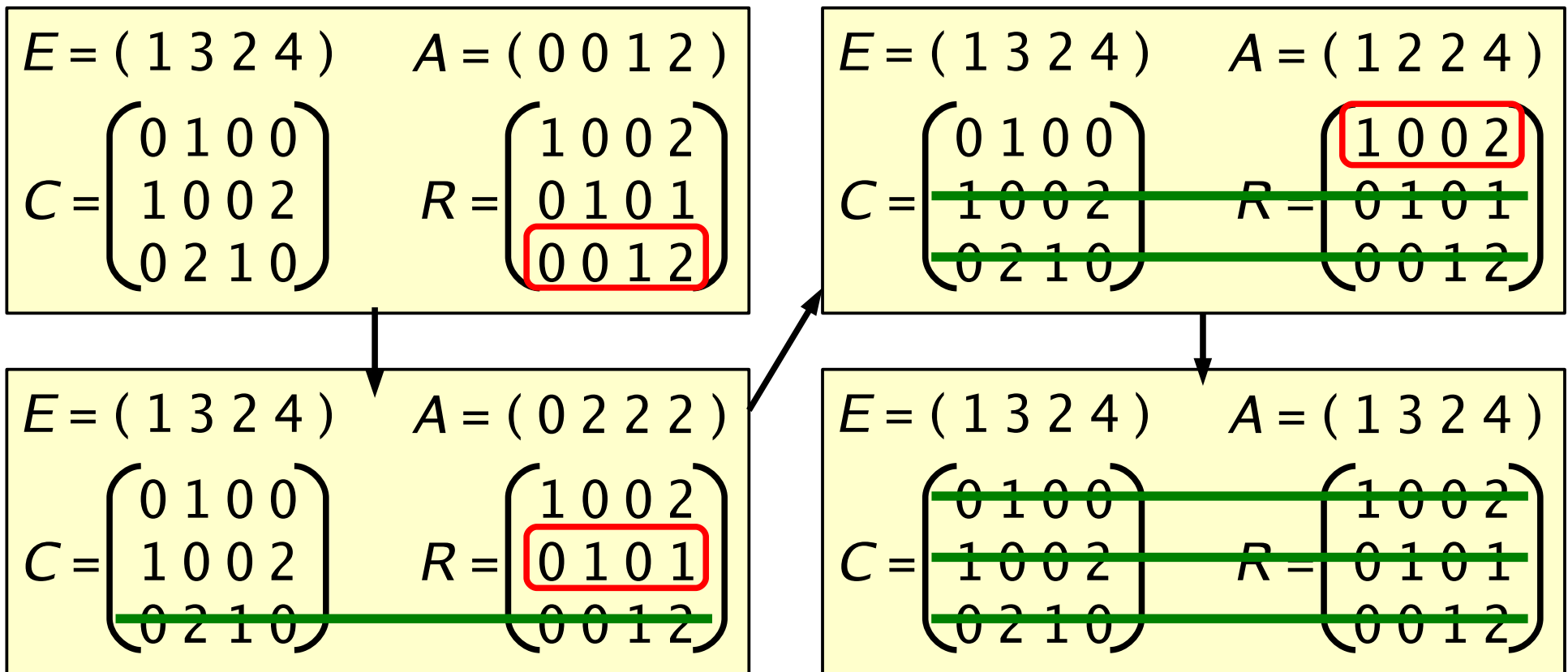
# Deadlocks (4)

- Deadlock-Erkennung:

		ISDN-Karte	Streamer	Scanner	DVD-Brenner	
$E = ( 1 \ 3 \ 2 \ 4 )$						
Ressourcenvektor						
$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix}$						
Belegungsmatrix						
$A = ( 0 \ 0 \ 1 \ 2 )$						
Ressourcenrestvektor						
$R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$						
Anforderungsmatrix						
						Prozess 1
						Prozess 2
						Prozess 3

# Deadlocks (5)

- 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)

- zusammenhängende Speicherzuteilung
  - Buddy-System
  - Segmentierung
- nicht-zshgd. Speicherzuteilung: Paging
  - virtuelle vs. physische Adressen, Seiten vs. Rahmen
  - (ein-/mehrstuf.) Seitentabellen, Adressübersetzung
  - Lokalität, TLB (Translation Look-aside Buffer)

Paging mit folgenden Parametern:

- 32-Bit-Adressbus
- 16 KB Seitengröße
- 2 GB RAM
- 3-stufiges Paging

Zu berechnen:

- a) maximale Anzahl der adressierbaren virtuellen Seiten
- b) Größe der Seitentabelle(n)
- c) Anzahl der Tabellen

- a) 16 KB (Seitengröße) =  $2^4 \times 2^{10}$  Byte  
=  $2^{14}$  Byte,  
d.h.: Offset ist 14 Bit lang



Seitennummer

Offset

Also gibt es  $2^{18}$  virtuelle Seiten

b) Zur Seitentabelle:

In 2 GB RAM passen  $2 \text{ G} / 16 \text{ K}$   
=  $128 \text{ K} = 2^{17}$  Seitenrahmen

Ein Eintrag in der Seitentabelle benötigt  
darum 17 Bit, in der Praxis 4 Byte.

→ Platzbedarf **einer** Tabelle:

$$\begin{aligned} &\#(\text{Einträge}) \times \text{Größe}(\text{Eintrag}) \\ &= 2^6 \times 4 \text{ Byte} = 2^8 \text{ Byte} = 256 \text{ Byte} \end{aligned}$$

Es gibt 1 äußere,  $2^6$  mittlere und  $2^{12}$   
innere Seitentabellen

- Partitionen (klassisch: primär, erweitert, logisch)
- Linux-Namen für Partititionen (/dev/hda1 etc.)
- partitionieren (fdisk)
- formatieren (mkfs.\*)
- mounten (mount, umount, /etc/fstab, Optionen)
- FS-Typen, Swap

