Betriebssysteme (BS)

VL 11.1 – Fadensynchronisation – Ebenenmodell

Volkmar Sieh / Daniel Lohmann

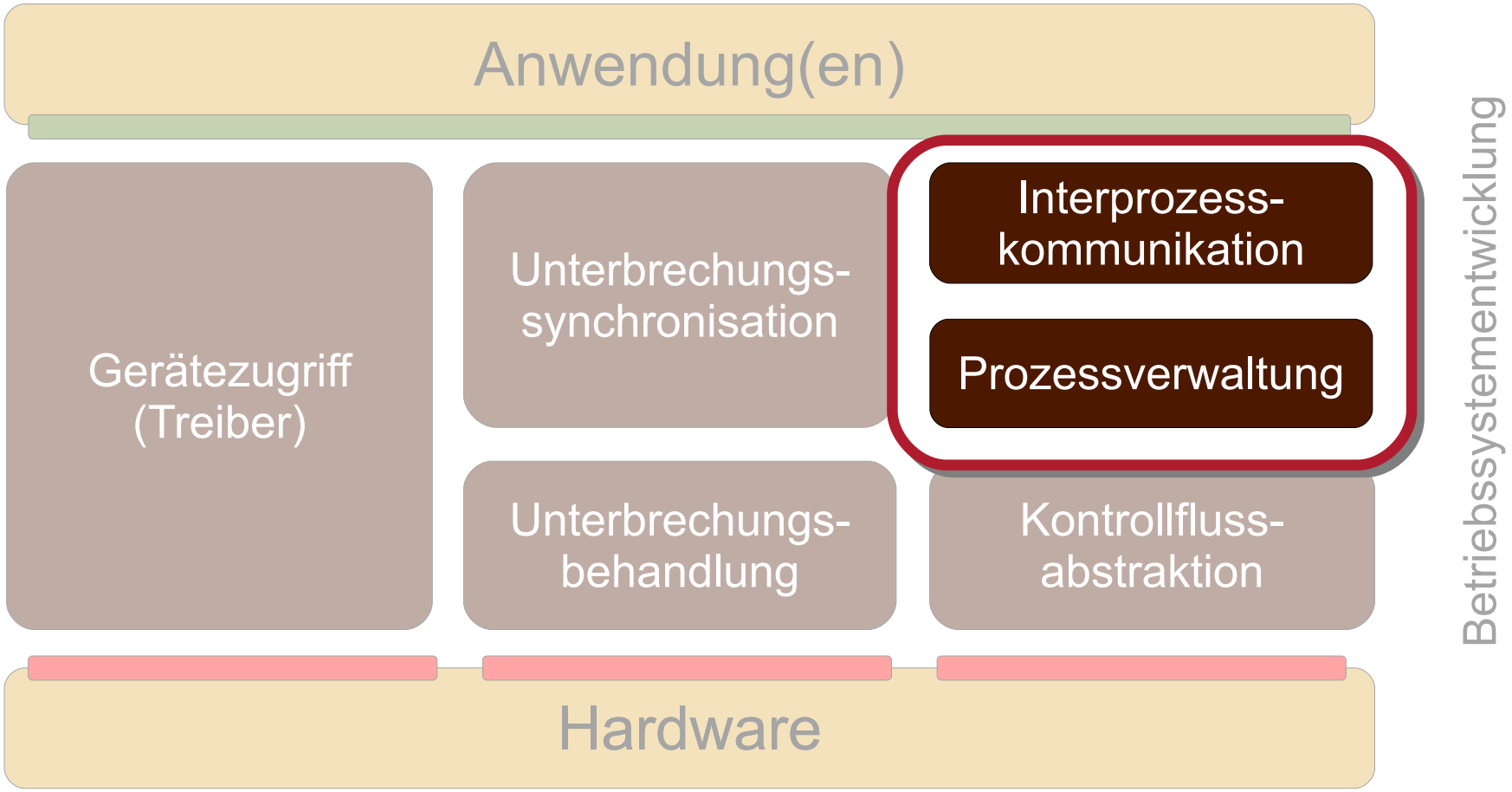
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 21 – 17. Januar 2022

https://www4.cs.fau.de/Lehre/WS21/V_BS

Überblick: Einordnung dieser VL

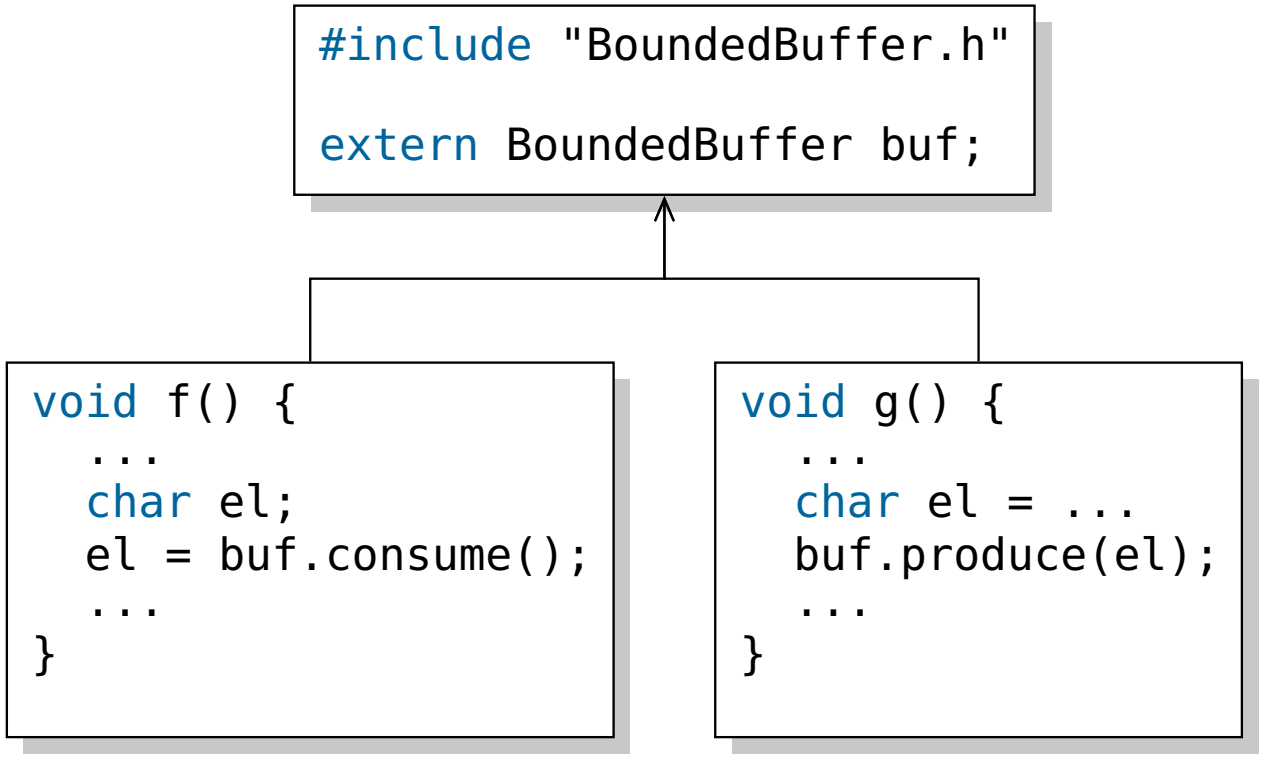


Agenda

- Einleitung
  - Motivation
  - Erstes Fazit
- Prioritätsebenenmodell mit Fäden
- Mechanismen
  - Randbedingungen
  - Mutex, Implementierungsvarianten
  - Passives Warten
  - Semaphore
- Beispiel: Windows
  - Warteobjekte
  - Optimierungen für Mehrkernsysteme
- Zusammenfassung
- Referenzen

Motivation Szenario

- Gegeben: Programmfäden <f> und <g>
  - Präemtives Scheduling (z. B. *round robin*)
  - Zugriff auf gemeinsame Datenstruktur `buf`



Motivation Szenario

- Gegeben: Programmfäden <f> und <g>
  - **Problem:** Pufferzugriffe können überlappen

```
char BoundedBuffer::consume() {
  int elements = occupied;
  if (elements == 0) return 0;
  char result = buf[nextout];
  nextout++; nextout %= SIZE;
```

↯ resume <g>

```
...
void BoundedBuffer::produce(char data) {
  int elements = occupied;
  if (elements == SIZE) return;
  buf[nextin] = data;
  nextin++; nextin %= SIZE;
  occupied = elements + 1;
}
...
```

↯ resume <f>

```
  occupied = elements - 1;
  return result;
}
```

Motivation Szenario

- Gegeben: Programmfäden <f> und <g>
  - **Problem:** Pufferzugriffe können überlappen

```
char BoundedBuffer::consume() {
  int elements = occupied;
  if (elements == 0) return 0;
  char result = buf[nextout];
  nextout++; nextout %= SIZE;
```

↯ resume <g>

```
...
void BoundedBuffer::produce(char data) {
  int elements = occupied;
  if (elements == SIZE) return;
  buf[nextin] = data;
  nextin++; nextin %= SIZE;
  occupied = elements + 1;
}
...
```

↯ resume <f>

```
  occupied = elements - 1;
  return result;
}
```

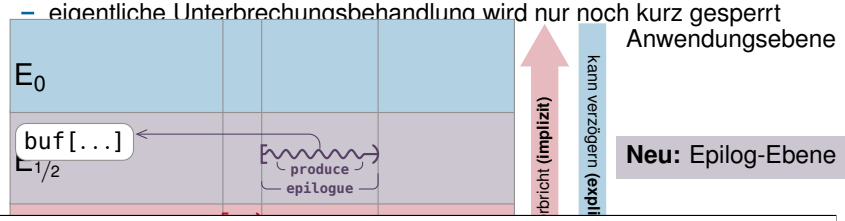
Das hatten wir doch schon mal...

Rückblick: VL 6 – Unterbrechungssynchronisation

Was ist diesmal anders?

Prolog/Epilog-Modell – Ansatz

- **Ansatz:** Latenzverbergung durch zusätzliche Ebene
  - Wir fügen eine weitere *logische* Ebene ein: $E_{1/2}$
    - $E_{1/2}$ liegt zwischen der Anwendungsebene E_0 und den UB-Ebenen $E_{1\ldots n}$
  - Unterbrechungsbehandlung wird zweigeteilt in Prolog und Epilog
    - Prolog arbeitet auf Unterbrechungsebene $E_{1\ldots n}$
    - Epilog arbeitet auf der neuen (Software-)Ebene $E_{1/2}$ (Epilogebene)
  - Zustand liegt (so weit wie möglich) auf der Epilogebene
    - eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt

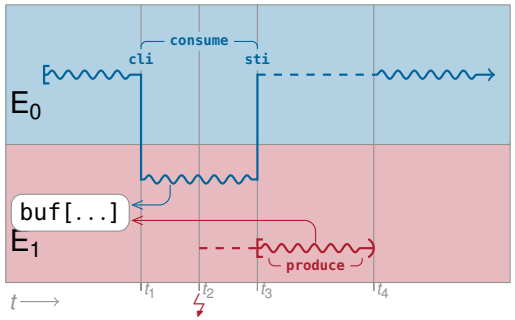


Bounded Buffer – Lösung mit harter Synchronisation

Zugriff „von oben“ wird hart synchronisiert: Für die Ausführung von `consume()` wechselt der Kontrollfluss auf E_1

```
char consume() {
  cli();
  ...
  char result = buf[nextout++];
  ...
  sti();
  return result;
}
```

```
void produce(char data) {
  // hier nichts zu tun
  ...
  buf[nextin++] = data;
  ...
  //hier nichts zu tun
}
```



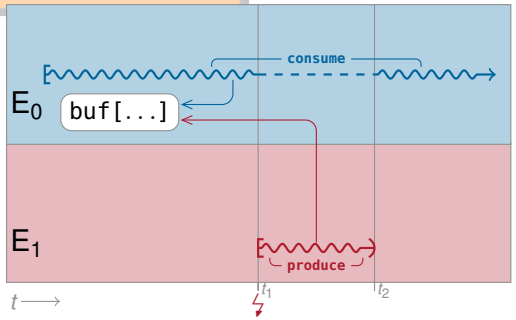
Zustand liegt (logisch) auf E_1

Bounded Buffer – Ansatz mit weicher Synchronisation

Zugriff „von unten“ wird weich synchronisiert: `consume()` liefert ein korrektes Ergebnis, auch wenn während der Abarbeitung `produce()` ausgeführt wurde.

```
char consume() {
  ?
}
```

```
void produce(char data) {
  ?
}
```



Zustand liegt (logisch) auf E_0

Erstes Fazit

- **Bisher:** Konsistenzsicherung bei Zugriffen von Kontrollflüssen aus verschiedenen Ebenen
  - Zustand wurde auf einer Ebene „platziert”
  - Sicherung entweder „von oben” (hart) oder „von unten” (weich)
  - Innerhalb einer Ebene wurde implizit sequentialisiert
- **Nun:** Konsistenzsicherung bei Zugriffen von Kontrollflüssen aus derselben Ebene
  - Fäden können jederzeit durch andere Fäden verdrängt werden
  - Fäden können echt parallel arbeiteten (bei mehreren CPUs)

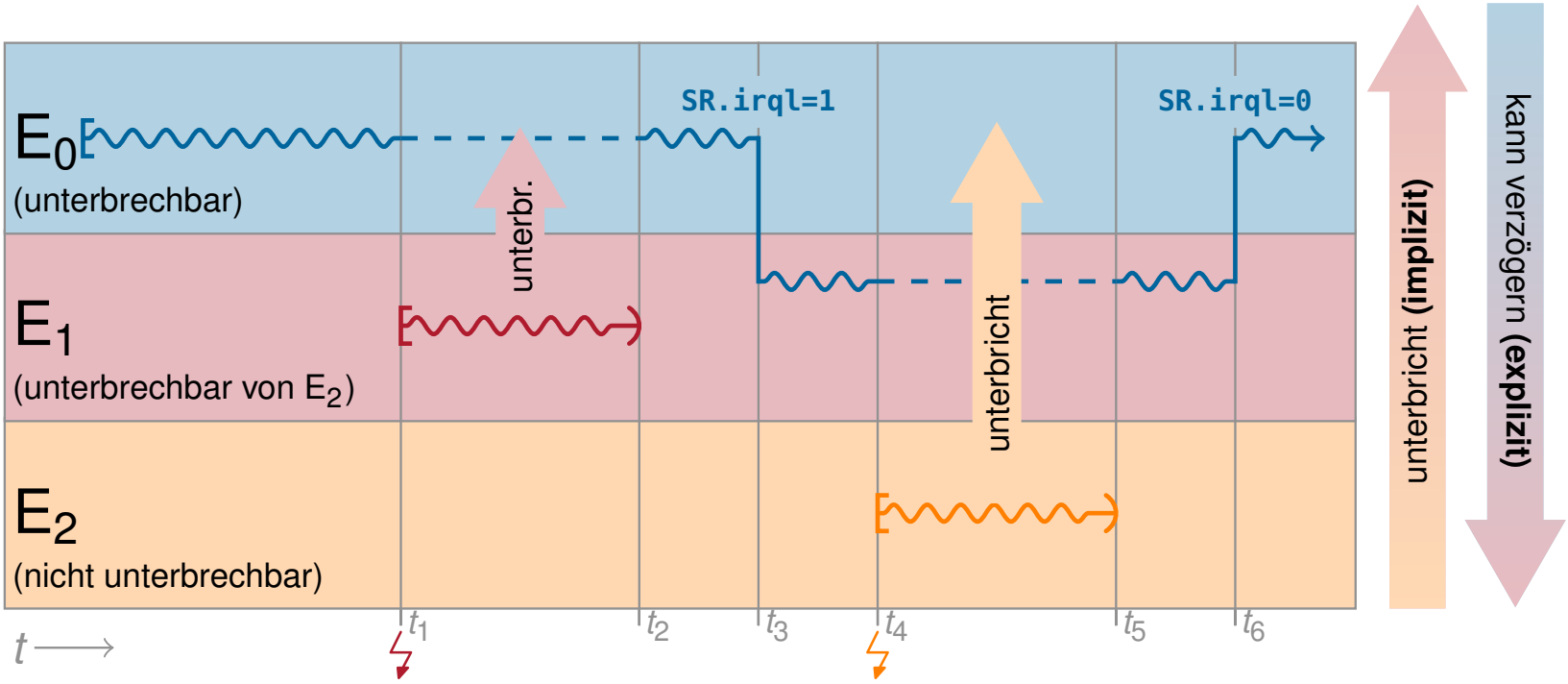
Das ist ja auch der Sinn von Fäden!

Agenda

Prioritätsebenenmodell

(Wiederholung von VL 6)

- Kontrollflüsse auf E_l werden
  1. jederzeit unterbrochen durch Kontrollflüsse von E_m (für $m > l$)
  2. nie unterbrochen durch Kontrollflüsse von E_k (für $k \leq l$)
  3. sequentialisiert mit weiteren Kontrollflüssen von E_l



Prioritätsebenenmodell (Wiederholung von VL 6)

- Kontrollflüsse auf E_l werden
  1. jederzeit unterbrochen durch Kontrollflüsse von E_m (für $m > l$)
  2. nie unterbrochen durch Kontrollflüsse von E_k (für $k \leq l$)
  3. sequentialisiert mit weiteren Kontrollflüssen von E_l

Das ist der Knackpunkt!

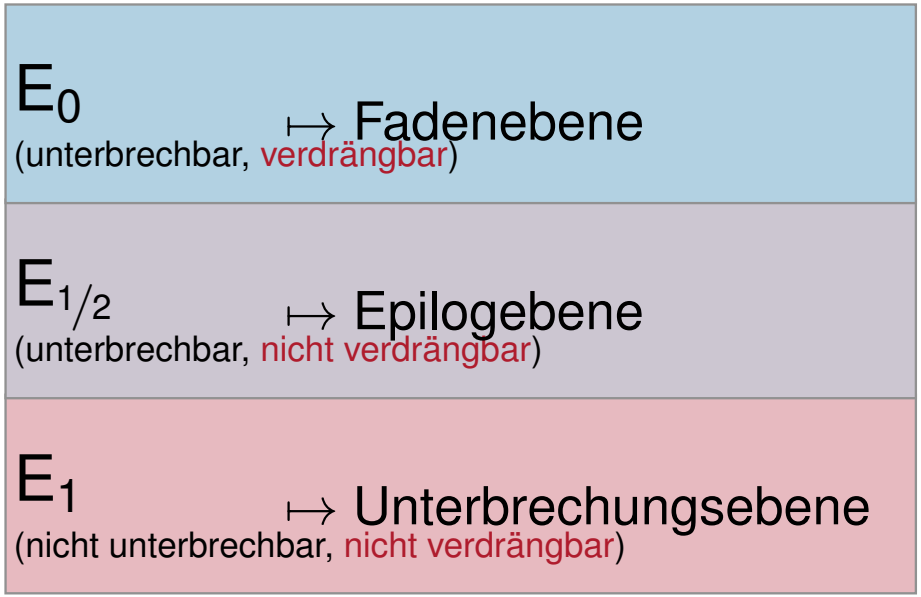
Mit der Unterstützung präemptiver Fäden können wir Annahme 3 nicht länger aufrechterhalten:

- keine *run-to-completion*–Semantik mehr
- Zugriff auf geteilten Zustand nicht mehr implizit sequentialisiert

Dies gilt für alle Ebenen, die Verdrängung (*Preemption*) oder echte Parallelität von Kontrollflüssen erlauben, also insbesondere die Anwendungsebene E_0.

Erweitertes Prioritätsebenenmodell

- Kontrollflüsse auf E_l werden
  1. jederzeit unterbrochen durch Kontrollflüsse von E_m (für $m > l$)
  2. nie unterbrochen durch Kontrollflüsse von E_k (für $k \leq l$)
  3. jederzeit verdrängt durch Kontrollflüsse von E_l (für $l = 0$)



Kontrollflüsse der E_0 (Fadenebene) sind **verdrängbar**.

Für die Konsistenzssicherung auf dieser Ebene brauchen wir zusätzliche **Mechanismen** zur **Fadensynchronisation**.