# Betriebssysteme (BS)

## VL 6.1 – Unterbrechungen, Synchronisation – Motivation

**Volkmar Sieh** / **Daniel Lohmann**

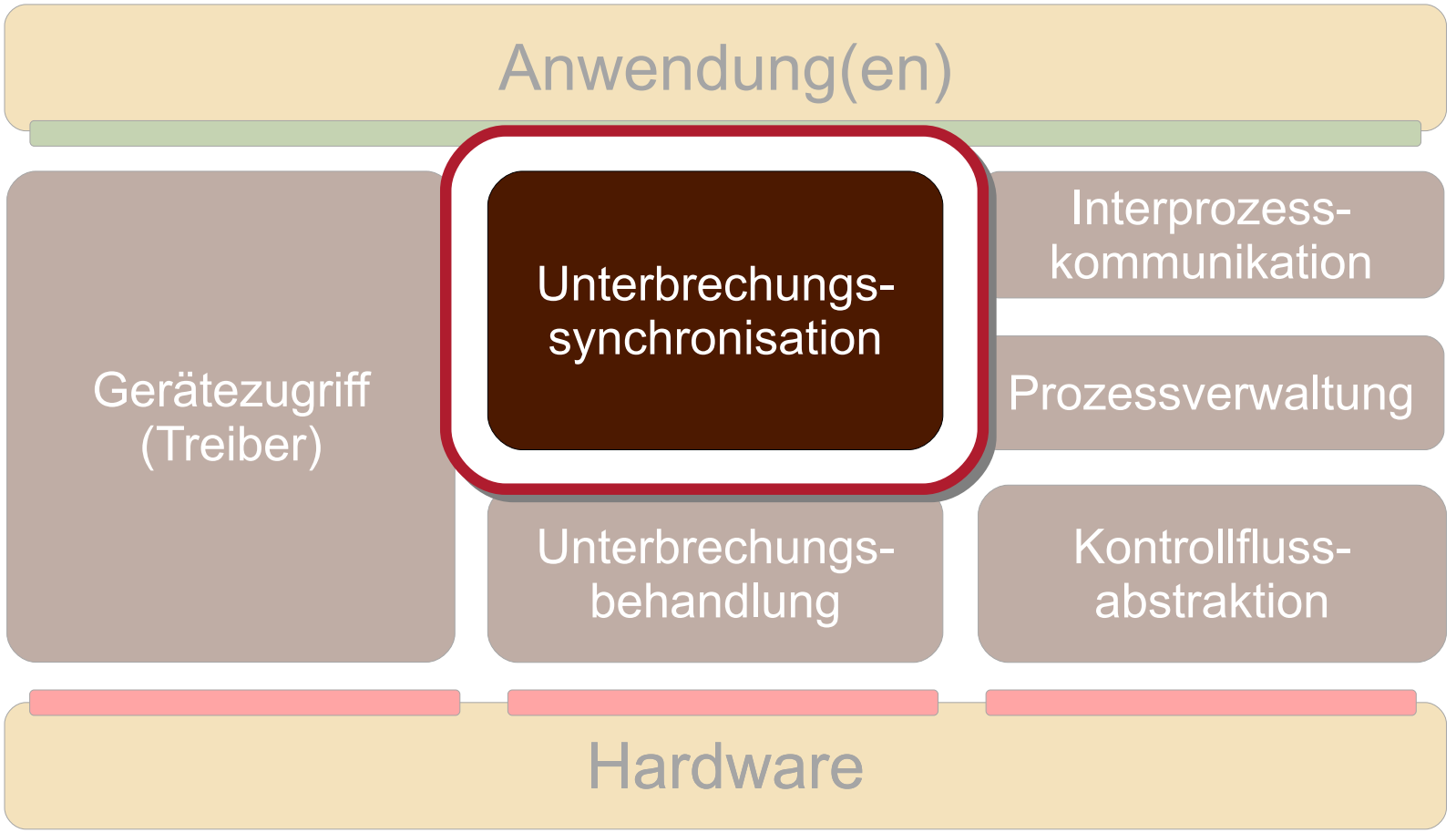
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 20 – 30. November 2020

https://www4.cs.fau.de/Lehre/WS20/V_BS

# Überblick: Einordnung dieser VL




Betriebssystementwicklung

# Agenda

Einleitung
Prioritätsebenenmodell
Harte Synchronisation
Weiche Synchronisation
Prolog/Epilog-Modell
Zusammenfassung
Referenzen

# Agenda



# Motivation: Konsistenzprobleme

## Zustandsänderungen ...

- sind Sinn und Zweck der Unterbrechungsbehandlung
  - Gerätetreiber müssen über den Abschluss einer E/A Operation informiert werden
  - der Scheduler muss erfahren, dass eine Zeitscheibe abgelaufen ist
- müssen mit Vorsicht durchgeführt werden
  - Unterbrechungen können zu jeder Zeit auftreten
  - kritisch sind Daten/Datenstrukturen, die der normale Kontrollfluss und die Unterbrechungsbehandlung sich teilen



Beispiele aus der letzten Vorlesung

## Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...

```
// Pufferklasse in C++
class BoundedBuffer {
  char buf[SIZE]; int occupied; int nextin, nextout;
public:
  BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
  void produce(char data) {        // Unterbrechungsbehandlung:
    int elements = occupied;       // Elementzähler merken
    if (elements == SIZE) return; // Element verloren
    buf[nextin] = data;            // Element schreiben
    nextin++; nextin %= SIZE;      // Zeiger weitersetzen
    occupied = elements + 1;       // Zähler erhöhen
  }
  char consume() {                 // normaler Kontrollfluss:
    int elements = occupied;       // Elementzähler merken
    if (elements == 0) return 0;  // Puffer leer, kein Ergebnis
    char result = buf[nextout];    // Element lesen
    nextout++; nextout %= SIZE;    // Lesezeiger weitersetzen
    occupied = elements - 1;       // Zähler erniedrigen
    return result;                 // Ergebnis zurückliefern
} };
```

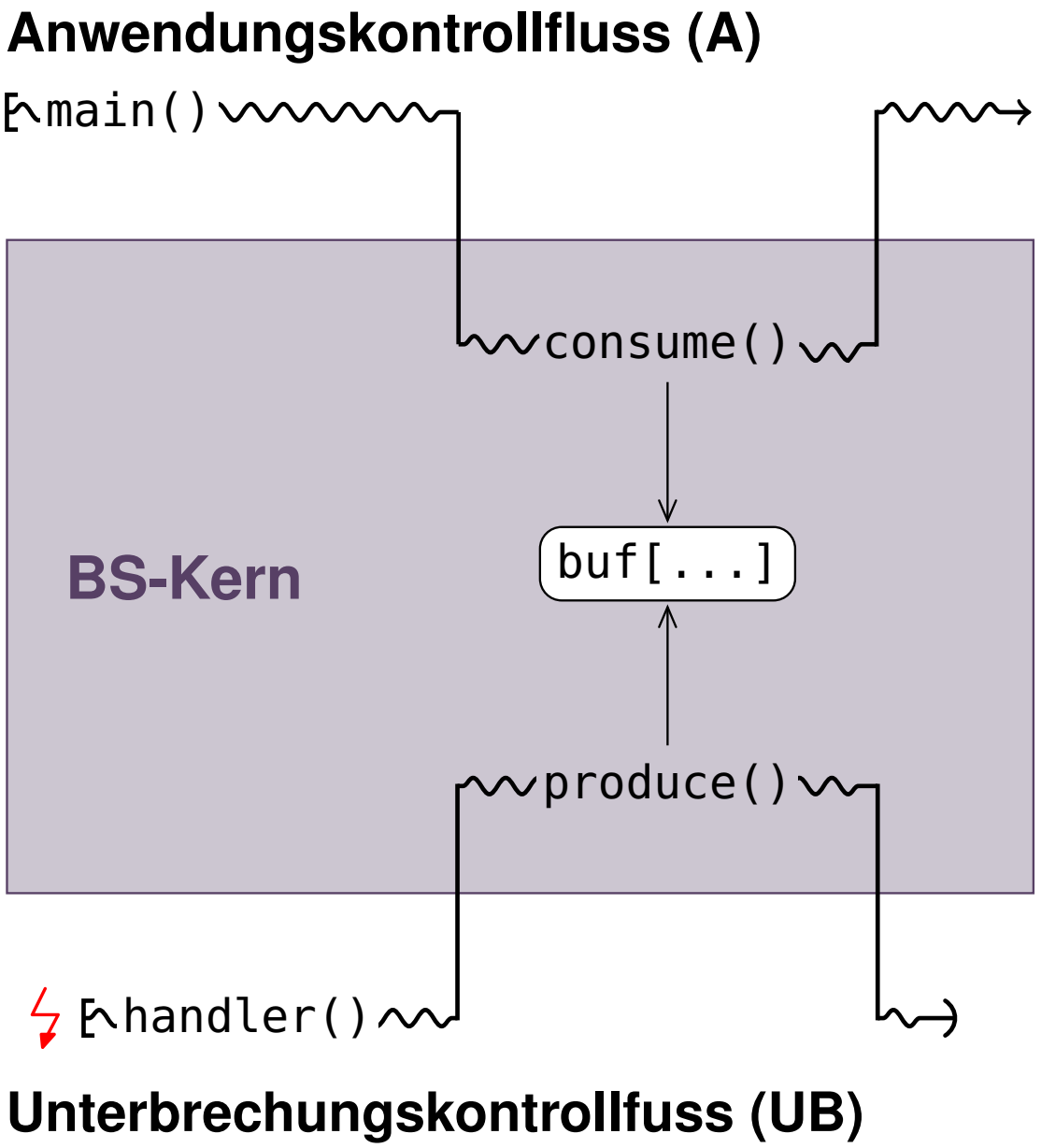


# Motivation: Ursache

Kontrollflüsse
“von oben”

“begegnen”
sich im Kern

und “von unten”

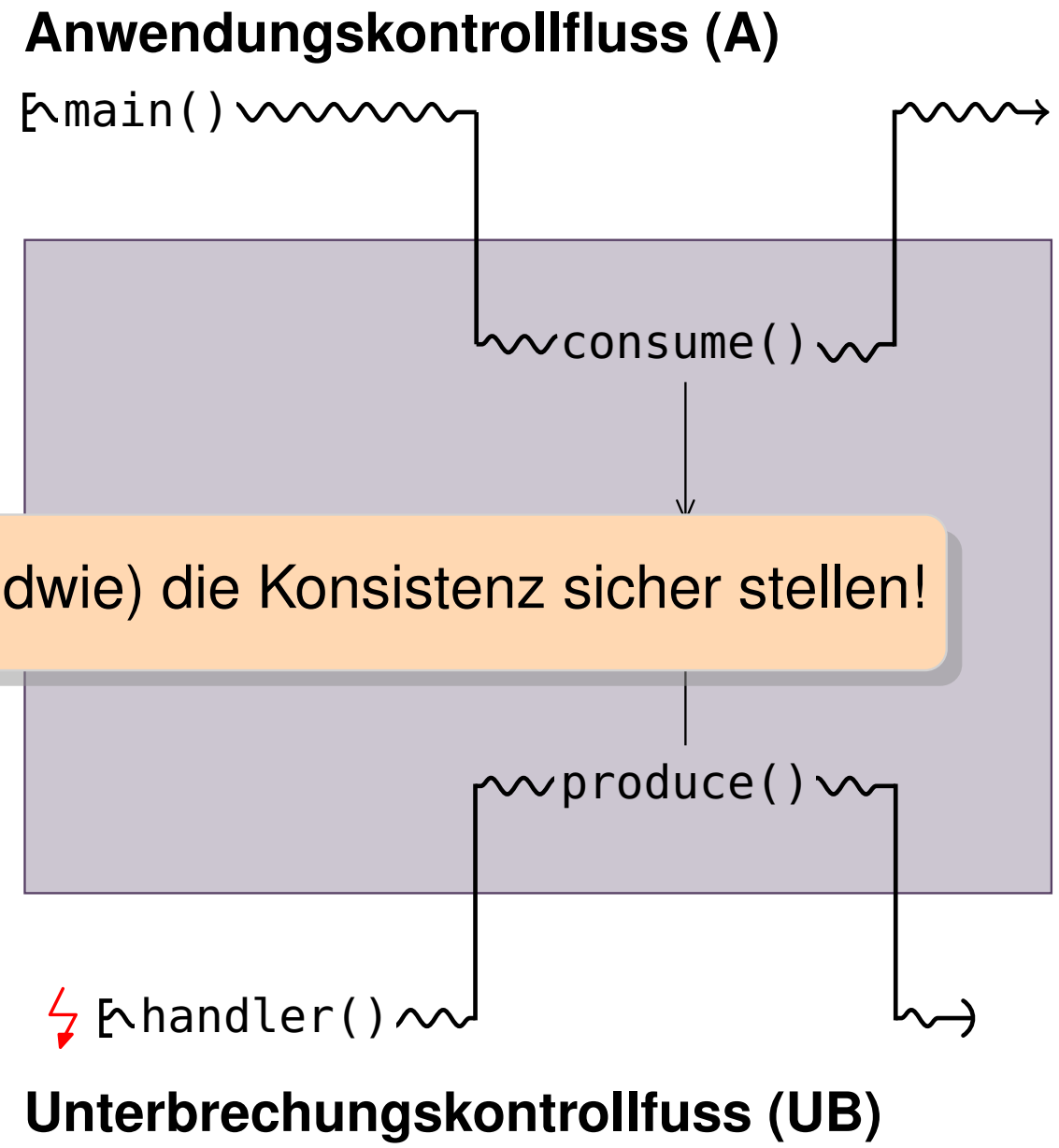



# Motivation: Ursache

Kontrollflüsse
“von oben”

“b
si

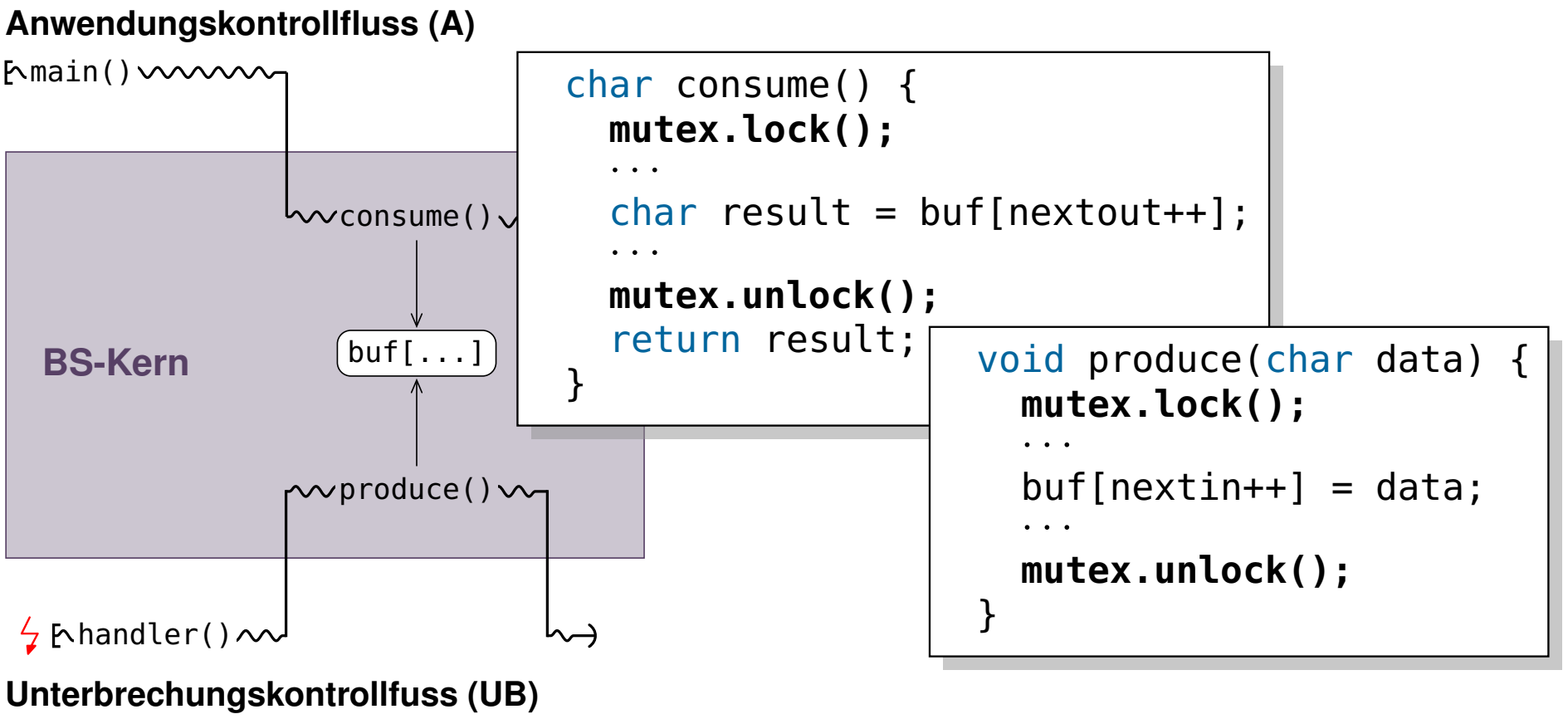
und “von unten”




Wir müssen (irgendwie) die Konsistenz sicher stellen!

# Naiver Lösungsansatz

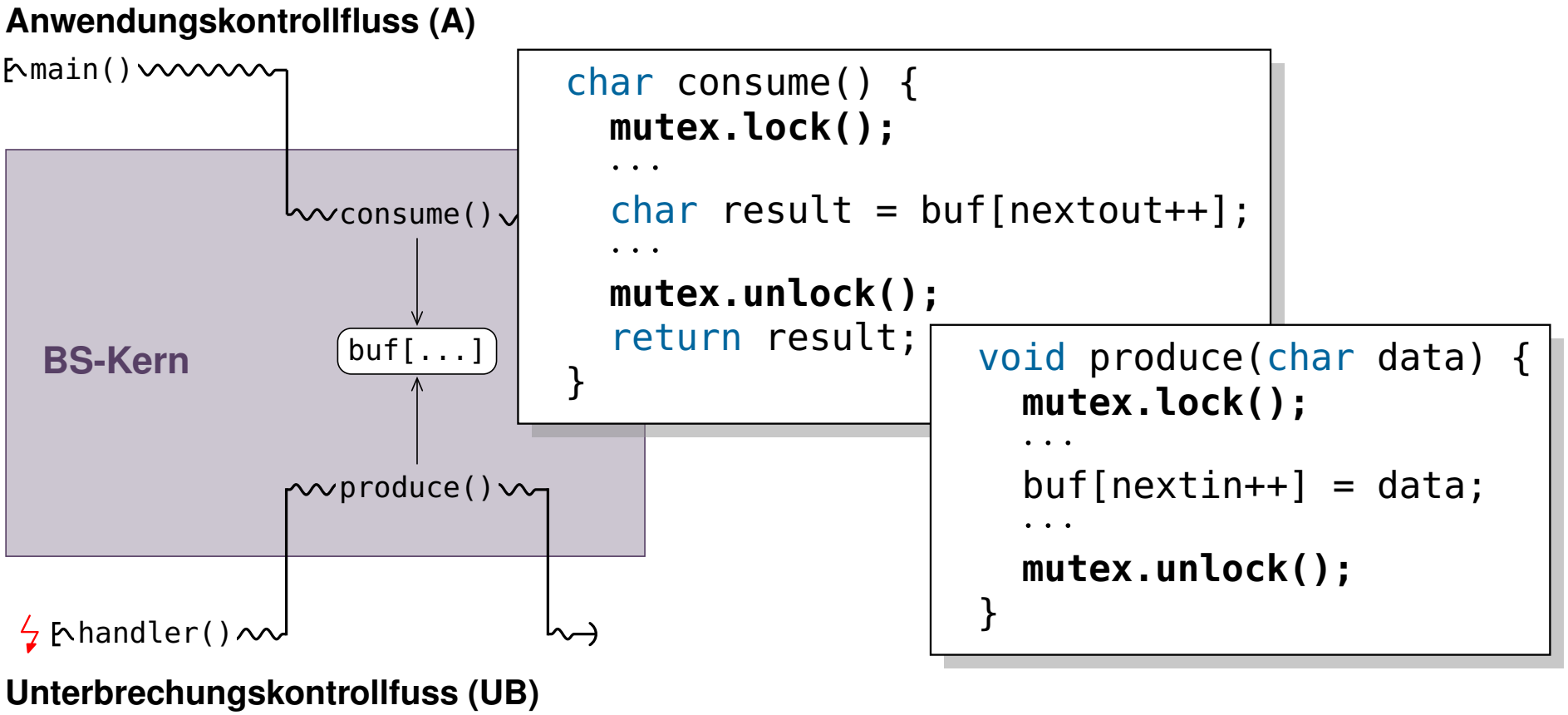
- Zweiseitige Synchronisation
  - gegenseitiger Ausschluss durch Mutex, *Spin-Lock*, ... (vgl. [SP])
  - wie zwischen zwei Prozessen



# Naiver Lösungsansatz

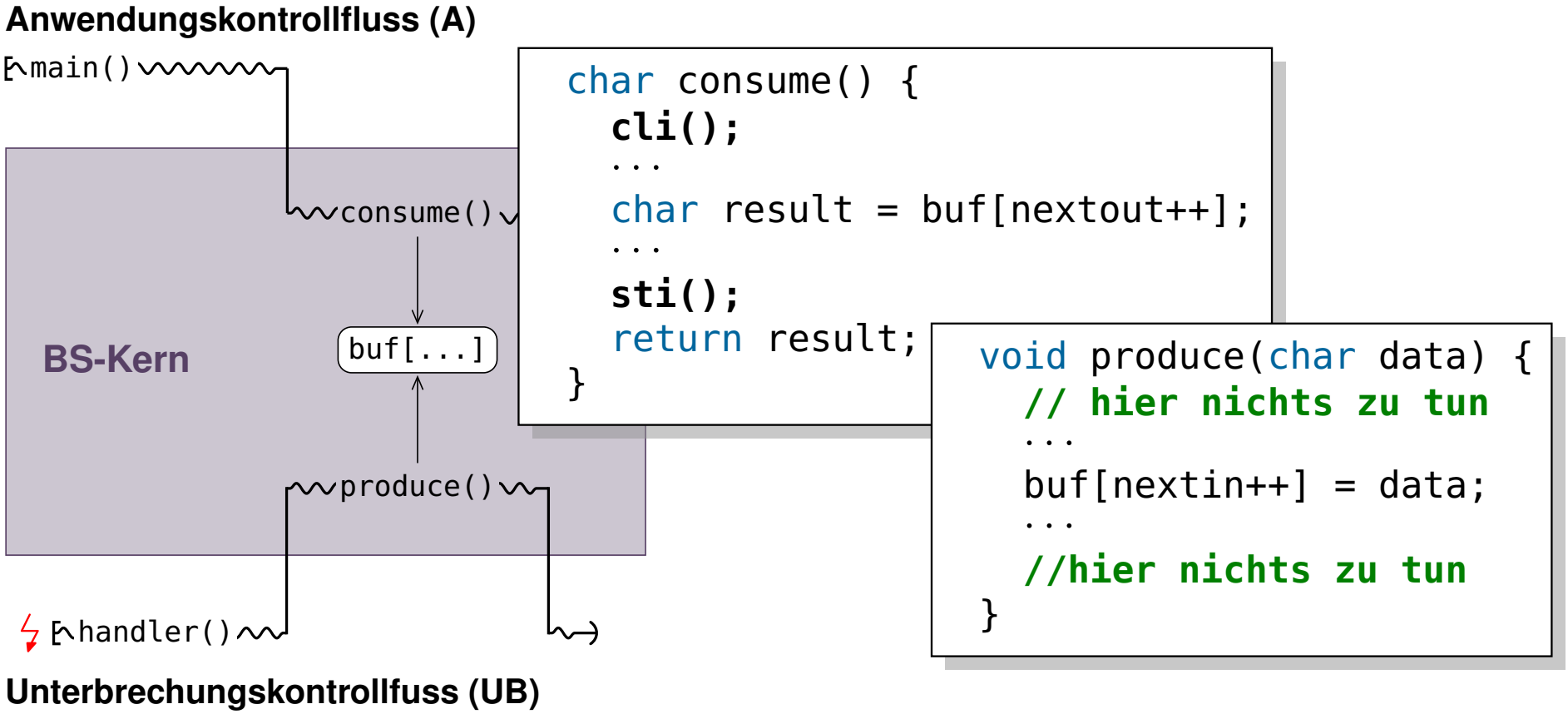
- Zweiseitige Synchronisation
  - gegenseitiger Ausschluss durch M
  - wie zwischen zwei Prozessen

Zweiseitige Synchronisation funktioniert **natürlich nicht!**



# Besserer Lösungsansatz

- Einseitige Synchronisation
  - Unterdrückung der Unterbrechungsbehandlung im Verbraucher
  - Operationen `disable_interrupts()` `enable_interrupts()`
    (im Folgenden o. B. d. A. in „Intel“-Schreibweise: `cli()` / `sti()`)

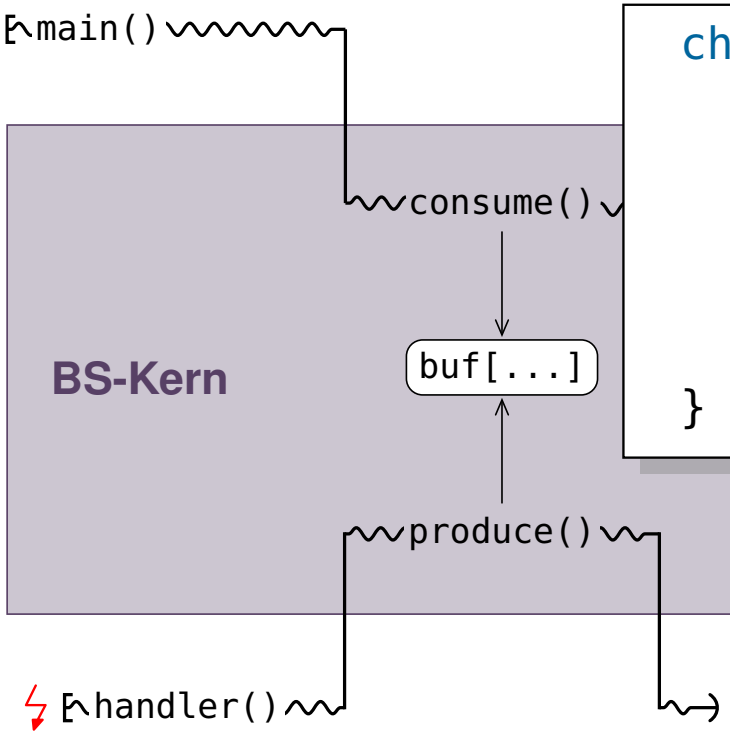


# Besserer Lösungsansatz

- Einseitige Synchronisation
  - Unterdrückung der Unterbrechungs
  - Operationen `disable_interrupts(`
    (im Folgenden o. B. d. A. in „Intel“-Schreibw

Einseitige Synchronisation funktioniert. [Warum?]

**Anwendungskontrollfluss (A)**




```
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
}
```

**Unterbrechungskontrollfuss (UB)**

# Erstes Fazit

- Konsistenzsicherung zwischen
  - Anwendungskontrollfluss (A) und
  - Unterbrechungsbehandlung (UB)

  muss anders erfolgen als zwischen Prozessen
- Die Beziehung zwischen A und UB ist **asymmetrisch**
  - Es handelt sich um „verschiedene Arten“ von Kontrollflüssen
  - UB *unterbricht* Anwendungskontrollfluss
    - implizit, an beliebiger Stelle
    - hat immer Priorität, läuft durch (*run-to-completion*)
  - A kann UB *unterdrücken* (besser: *verzögern*)
    - explizit, mit `cli` / `sti` (Grundannahme 5 aus VL 4)
- Synchronisation / Konsistenzsicherung erfolgt **einseitig**

# Erstes Fazit

- Konsistenzsicherung zwischen
  - Anwendungskontrollfluss (A) und
  - Unterbrechungsbehandlung (UB)

  muss anders erfolgen als zwischen Prozessen
- Die Beziehung zwischen A und UB ist **asymmetrisch**
  - Es handelt sich um „verschiedene Arten“ von Kontrollflüssen
  - UB *unterbricht* Anwendungskontrollfluss
    - implizit, an beliebiger Stelle
    - hat immer Priorität, läuft durch (*run-to-completion*)
  - A kann UB *unterdrücken* (besser: *verzögern*)
    - explizit, mit `cli` / `sti` (Grundannahme 5 aus VL 4)
- Synchronisation / Konsistenzsicherung erfolgt **einseitig**

Diese Tatsachen müssen wir **beachten**!

(Das heißt aber auch: Wir können sie **ausnutzen**)