Betriebssysteme (BS)

VL 6.3 – Unterbrechungen, Synchronisation – Harte/weiche Synchronisation

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 20 – 30. November 2020

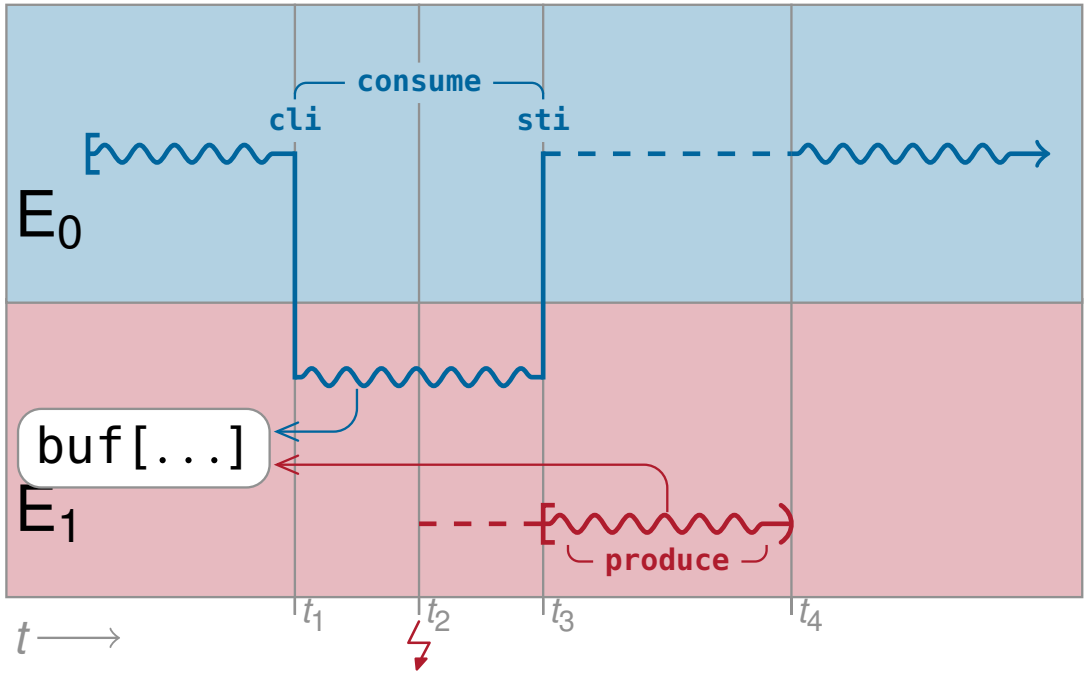
Agenda

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

Harte Synchronisation: Bewertung

- **Vorteile**
  - Konsistenz ist sicher gestellt
    - auch bei komplexen Datenstrukturen und Zugriffsmustern
    - unabhängig davon, was der Compiler macht
  - einfach anzuwenden, „funktioniert immer“
    - im Zweifelsfall legt man einfach sämtlichen Zustand auf die höchstpriore Ebene
- **Nachteile**
  - Breitbandwirkung
    - Es werden pauschal alle Unterbrechungsbehandlungen (Kontrollflüsse) auf und unterhalb der Zustandsebene verzögert
  - Prioritätsverletzung
    - Es werden Kontrollflüsse höherer Priorität verzögert
  - prophylaktisches Verfahren
    - Nachteile werden in Kauf genommen, obwohl die Wahrscheinlichkeit, dass tatsächlich eine relevante Unterbrechung eintrifft, sehr klein ist.

Harte Synchronisation: Bewertung (Forts.)

- Ob die Nachteile erheblich sind, hängt ab von
  - Häufigkeit,
  - durchschnittlicher Dauer,
  - maximaler Dauer

  der Verzögerung.
- Kritisch ist vor allem die **maximale Dauer**
  - hat direkten Einfluss auf die anzunehmende Latenz
  - Wird die Latenz zu hoch, können Daten verloren gehen
    - *edge-triggered* Unterbrechungen gehen verloren
    - Daten werden zu langsam von EA-Gerät abgeholt

Fazit

Harte Synchronisation ist eher **ungeeignet** für die Konsistenzsicherung **komplexer Datenstrukturen**

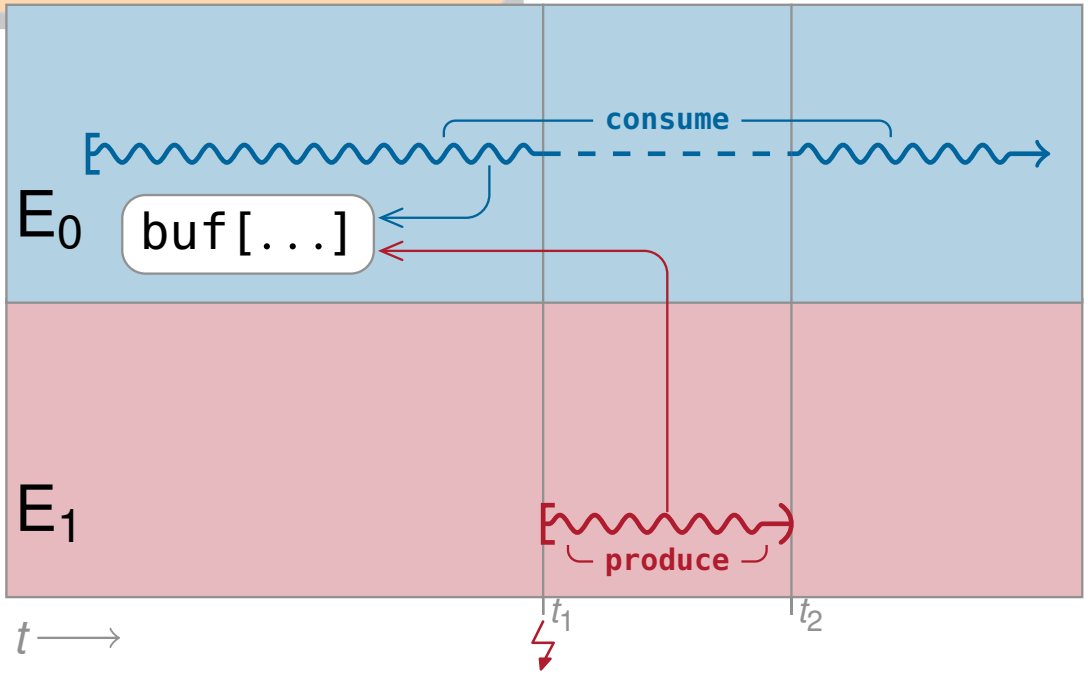
Agenda

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() {

      ?

}
```

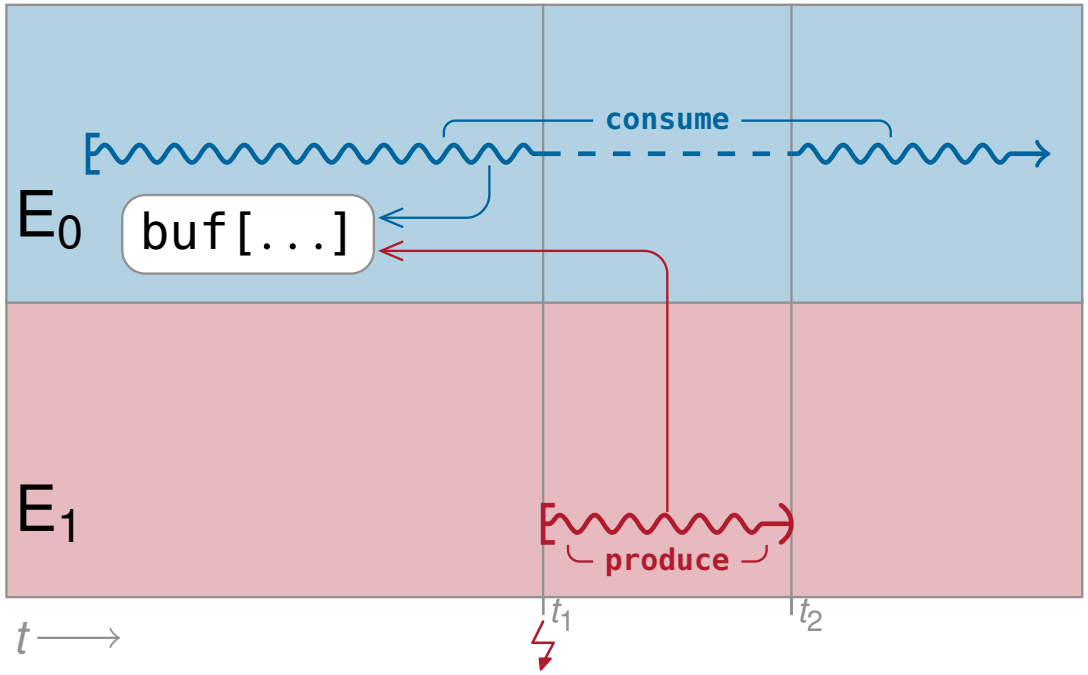
```
void produce(char data) {

      ?

}
```



Zustand liegt (logisch) auf E_0

Bounded Buffer – Konsistenzbedingungen, Annahmen

- Konsistenzbedingung
  - Ergebnis einer unterbrochenen Ausführung soll äquivalent sein zu dem einer sequentiellen Ausführung der Operation
    - *entweder* `consume()` vor `produce()` *oder* `consume()` nach `produce()`
- Annahmen
  - `produce()` unterbricht `consume()`
    - alle anderen Kombinationen kommen nicht vor
  - `produce()` läuft immer durch (*run-to-completion*)



Bounded Buffer – Implementierung aus der letzten VL

Kritisch ist der gemeinsam verwendete Zustand

```
// 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;       // Elementzaehler merken
    if (elements == SIZE) return; // Element verloren
    buf[nextin] = data;            // Element schreiben
    nextin++; nextin %= SIZE;      // Zeiger weitersetzen
    occupied = elements + 1;       // Zaehler erhoehen
  }
  char consume() {             // normaler Kontrollfluss:
    int elements = occupied;       // Elementzaehler merken
    if (elements == 0) return 0;  // Puffer leer, kein Ergebnis
    char result = buf[nextout];    // Element lesen
    nextout++; nextout %= SIZE;    // Lesezeiger weitersetzen
    occupied = elements - 1;       // Zaehler erniedrigen
    return result;                 // Ergebnis zurueckliefern
} };
```

Bounded Buffer – Implementierung aus der letzten VL

Kritisch ist der gemeinsam verwendete Zustand

Insbesondere Zustand,
auf den von beiden Seiten
schreibend zugegriffen wird.

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

Bounded Buffer – Alternative Implementierung

```
// Pufferklasse in C++ (alternativ)
class BoundedBuffer {
  char buf[SIZE]; int nextin, nextout;
public:
  BoundedBuffer(): nextin(0), nextout(0) {}
  void produce(char data) {

    if ((nextin + 1) % SIZE == nextout) return;
    buf[nextin] = data;
    nextin = (nextin + 1) % SIZE;

  }
  char consume() {

    if (nextout == nextin) return 0;
    char result = buf[nextout];
    nextout = (nextout + 1) % SIZE;

    return result;
} };
```

Diese alternative Implementierung kommt ohne gemeinsam beschriebenen Zustand aus.

Bounded Buffer – Alternative Implementierung

```
// Pufferklasse in C++ (alternativ)
class BoundedBuffer {
  char buf[SIZE]; int nextin, nextout;
public:
  BoundedBuffer(): nextin(0), nextout(0) {}
  void produce(char data) {

    if ((nextin + 1) % SIZE == nextout) return;
    buf[nextin] = data;
    nextin = (nextin + 1) % SIZE;

  }
  char consume() {

    if (nextout == nextin) return 0;
    char result = buf[nextout];
    nextout = (nextout + 1) % SIZE;

    return result;
} };
```

Allerdings gibt es hier jetzt Zustand, der von einer Seite gelesen und von der jeweils anderen beschrieben wird.

Bounded Buffer – Alternative Implementierung

```
// Pufferklasse in C++ (alternativ)
class BoundedBuffer {
  char buf[SIZE]; int nextin, nextout;
public:
  BoundedBuffer(): nextin(0), nextout(0) {}
  void produce(char data) {

    if ((nextin + 1) % SIZE == nextout) return;
    buf[nextin] = data;
    nextin = (nextin + 1) % SIZE;

  }
  char consume() {

    if (nextout == nextin) return 0;
    char result = buf[nextout];
    nextout = (nextout + 1) % SIZE;

    return result;
} };
```

Allerdings gibt es hier jetzt Zustand, der von einer Seite gelesen und von der jeweils anderen beschrieben wird.

An genau diesen Stellen müssen wir prüfen, ob die Konsistenzbedingung gilt.

Bounded Buffer – Analyse der neuen Implementierung

- Angenommen, die Unterbrechung von `consume()` erfolgt:
  - aus der Sicht von `consume()`
    - vor dem Lesen von **`nextin`** ⇔ `consume()` nach `produce()` ✔
    - nach dem Lesen von **`nextin`** ⇔ `consume()` vor `produce()` ✔
  - aus der Sicht von `produce()`
    - vor dem Schreiben von **`nextout`** ⇔ `produce()` vor `consume()` ✔
    - nach dem Schreiben von **`nextout`** ⇔ `produce()` nach `consume()` ✔

```
char consume() {
  if (nextout == nextin) return 0;
  char result = buf[nextout];
  nextout = (nextout + 1) % SIZE;
  return result;
}
```

Konsistenzbedingung ist in jedem Fall erfüllt!

```
void produce(char data) {
  if ((nextin + 1) % SIZE == nextout) return;
  buf[nextin] = data;
  nextin = (nextin + 1) % SIZE;
}
```

Systemzeit – Implementierung aus der letzten Vorlesung

```
/* globale Zeitvariable */
extern volatile time_t global_time;
```

```
/* Systemzeit abfragen */

  time_t time () {
    return global_time;
}
```

```
/* Unterbrechungs- *
 * behandlung      */
void timerHandler () {
  global_time++;
}
```

h8300-hms-g++ (16-Bit-Architektur)

```
time:
  mov global_time, %r0; lo
  mov global_time+2, %r1; hi
  ret
```

Problem:
Daten werden nicht atomar gelesen.

Systemzeit – Konsistenzbedingungen, Annahmen, Ansatz

- Konsistenzbedingung
  - Ergebnis einer unterbrochenen Ausführung soll äquivalent sein zu dem einer sequentiellen Ausführung der Operation
    - *entweder* `time()` vor `timerHandler()` *oder umgekehrt*
- Annahmen
  - `timerHandler()` unterbricht `time()`
    - alle anderen Kombinationen kommen nicht vor
  - `timerHandler()` läuft immer durch (*run-to-completion*)
- Lösungsansatz: In `time()` optimistisch herangehen

1. lese Daten unter der Annahme nicht unterbrochen zu werden
2. überprüfe, ob Annahme zutraf – wurden wir unterbrochen?
3. falls unterbrochen, setze neu auf ab Schritt 1

Systemzeit – Neue Implementierung

```
/* globale Zeitvariable */
extern volatile time_t global_time;
extern volatile bool interrupted;
```

```
/* Systemzeit abfragen */
time_t time () {
  time_t res;
  do {
    interrupted = false;
    res = global_time;
  } while (interrupted);
  return res;
}
```

```
/* Unterbrechungsbehandlung */
void timerHandler () {
  interrupted = true;
  global_time++;
}
```

Konsistenzbedingung ist nun in jedem Fall erfüllt!

Weiche Synchronisation: Bewertung

Vorteile

- Konsistenz ist sichergestellt (durch Unterbrechungstransparenz)
- Priorität wird nie verletzt
  - Kontrollflüsse der höherprioren Ebenen kommen immer durch
- Kosten entstehen entweder gar nicht oder nur im Konfliktfall
  - gar nicht ⇝ Beispiel Bounded Buffer
  - im Konfliktfall ⇝ optimistische Verfahren, Beispiel Systemzeit (zusätzliche Kosten durch Wiederaufsetzen)

Nachteile

- Lösungen häufig sehr komplex
  - Wenn man überhaupt eine Lösung findet, ist diese in der Regel schwer zu verstehen – und noch schwieriger zu verifizieren
- Lösungen häufig sehr fragil (bezüglich Randbedingungen)
  - Kleinste Änderungen können die Konsistenzgarantie zerstören
  - Codegenerierung des Compilers ist zu beachten
- Bei größeren Datenmengen steigen die Wiederaufsetzkosten

Weiche Synchronisation: Bewertung (Forts.)

Fazit

- Weiche Synchronisation durch Unterbrechungstransparenz ist **grundsätzlich erstrebenswert**!
- Es handelt sich bei den Algorithmen jedoch immer um **Speziallösungen** für **Spezialfälle**.
- Als allgemein verwendbares Mittel für die Sicherung **beliebiger Datenstrukturen** ist sie **nicht geeignet**.