Betriebssysteme (BS)

VL 8.1 – Koroutinen und Fäden – Einleitung

Volkmar Sieh / **Daniel Lohmann**

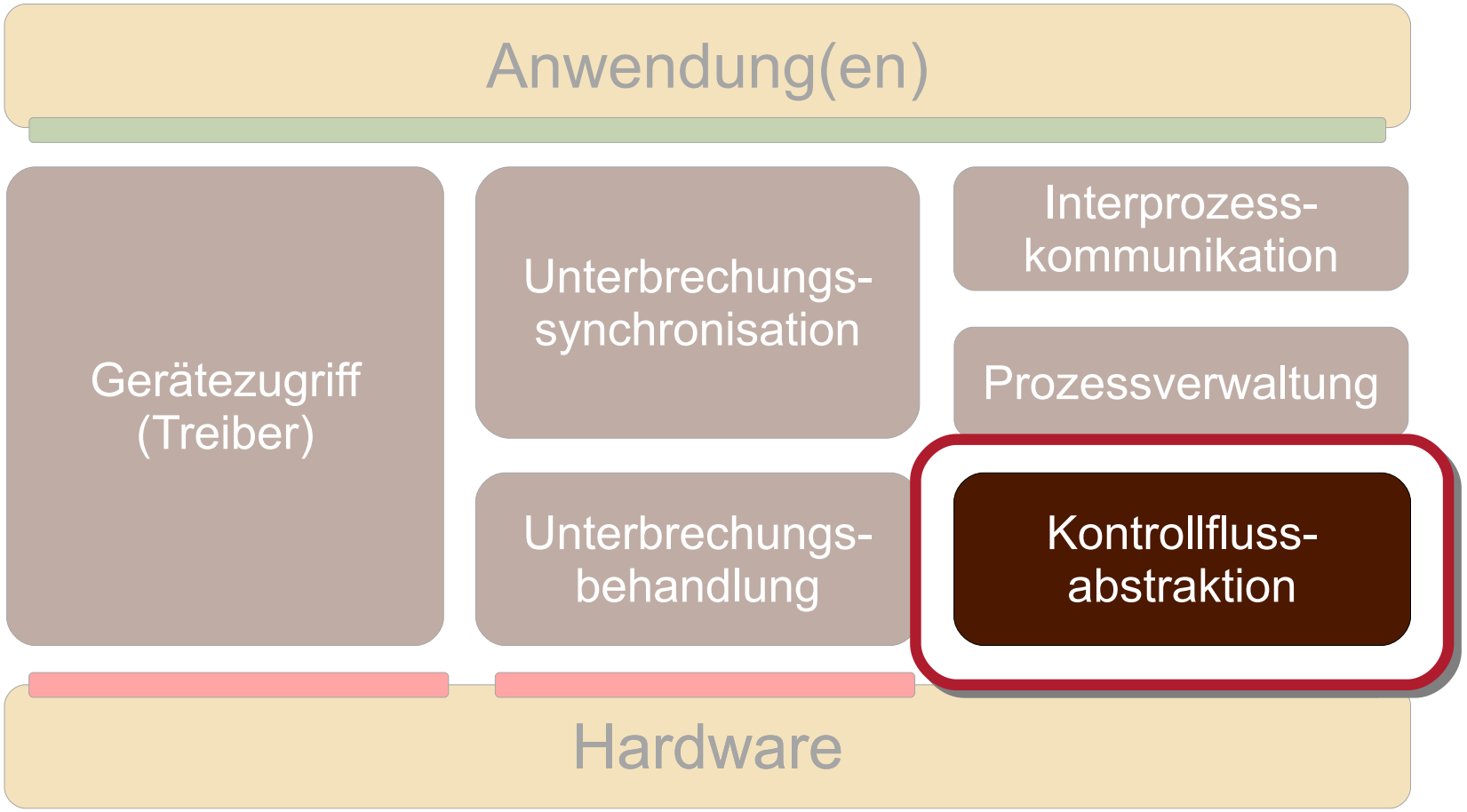
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 20 – 14. Dezember 2020

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

Überblick: Einordnung dieser VL



Agenda

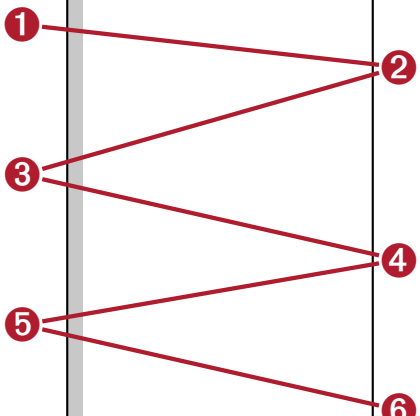
Motivation
Grundbegriffe
Implementierung
Ausblick
Zusammenfassung
Referenzen

Agenda

Motivation: Quasi-Parallelität

```
void f() {
    printf("f:1\n");
    printf("f:2\n");
    printf("f:3\n");
}
```

```
void g() {
    printf("g:A\n");
    printf("g:B\n");
    printf("g:C\n");
}
```




```
int main() {
    ?
}
```

- **Gegeben:** Funktionen `f()` und `g()`
- **Ziel:** `f()` und `g()` sollen „verschränkt” ablaufen

Im Folgenden einige Versuche...

Motivation: Quasi-Parallelität


```
void f() {
    printf("f:1\n");

    printf("f:2\n");

    printf("f:3\n");


}
```

```
void g() {

    printf("g:A\n");


    printf("g:B\n");

    printf("g:C\n");

}
```

```
int main() {

  f();
  g();
}
```

```
lohmann@faui48a>gcc routine.c -o routine
lohmann@faui48a>./routine
f:1
f:2
f:3
g:A
g:B
g:C
```

So funktioniert es natürlich nicht.

Motivation: Quasi-Parallelität


```
void f() {
    printf("f:1\n");
    g();

    printf("f:2\n");
    g();

    printf("f:3\n");
    g();


}
```

```
void g() {

    printf("g:A\n");



    printf("g:B\n");


    printf("g:C\n");

}
```

```
int main() {


  f();

}
```

```
lohmann@faui48a>gcc routine.c -o routine
lohmann@faui48a>./routine
f:1
g:A
g:B
g:C
f:2
...
```

So geht es wohl auch nicht.

Motivation: Quasi-Parallelität


```
void f() {
    printf("f:1\n");
    g();

    printf("f:2\n");
    g();

    printf("f:3\n");
    g();


}
```

```
void g() {

    printf("g:A\n");
    f();


    printf("g:B\n");
    f();

    printf("g:C\n");
    f();
}
```

```
int main() {


  f();

}
```

```
lohmann@faui48a>gcc routine.c -o routine
lohmann@faui48a>./routine
f:1
g:A
f:1
g:A
...
Segmentation fault
```

So **schon gar nicht!**

Motivation: Quasi-Parallelität


```
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;


}
```

```
void g_start() {

    printf("g:A\n");
    g = &&l1; goto *f;


l1: printf("g:B\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

```
void (*volatile f)();
void (*volatile g)();
int main() {
  f=f_start;
  g=g_start;
  f();

}
```

Und so?

Motivation: Quasi-Parallelität


```
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;


}
```

```
void g_start() {

    printf("g:A\n");
    g = &&l1; goto *f;


l1: printf("g:B\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

```
void (*volatile f)();
void (*volatile g)();
int main() {
  f=f_start;
  g=g_start;
  f();

}
```

```
lohmann@faui48a>gcc-2.95 -fomit-frame-
pointer -o coroutine coroutine.c
lohmann@faui48a>./coroutine
f:1
g:A
f:2
g:B
f:3
g:C
```

Klappt!

Motivation: Quasi-Parallelität


```
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;


}
```

```
void g_start() {

    printf("g:A\n");
    g = &&l1; goto *f;


l1: printf("g:B\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

```
void (*volatile f)();
void (*volatile g)();
int main() {
  f=f_start;
  g=g_start;
  f();

}
```

```
lohmann@faui48a>gcc-2.95 -fomit-frame-
pointer -o coroutine coroutine.c
lohmann@faui48a>./coroutine
f:1
g:A
f:2
g:B
f:3
g:C
```

Bitte nicht zu Hause nachmachen!

Quasi-Parallelität: Feststellungen

- C/C++ bietet keine Bordmittel für „verschränkte” Ausführung
  - einfache Funktionsaufrufe (Versuche 1 und 2)
    - laufen immer komplett durch (*run-to-completion*)
  - rekursive Funktionsaufrufe (Versuch 3)
    - dito, ⇝ Endlosrekursion und Stapelüberlauf
- Wir brauchen Systemunterstützung, um Kontrollflüsse „während der Ausführung” verlassen und wieder betreten zu können
  - ungefähr so wie in Versuch 4
    - „Fortsetzungs”-PC wird gespeichert, mit goto wieder aufgenommen
  - aber bitte ohne die damit einhergehenden Probleme!
    - *computed gotos* aus Funktionen sind undefiniert
    - Zustand besteht aus mehr als dem PC – was ist mit Registern, Stapel, . . .

Anmerkung: Aus Systemsicht („von unten”) würde der PC reichen!

- (PC) ⇔ *minimaler Kontrollflusszustand*
- alles weitere ist letztlich eine Entwurfsentscheidung des **Compilers** ⇝ [UE1]
- wird in der Praxis jedoch durch Hardwarehersteller nahegelegt (ISA, ABI)