Betriebssystem-Entwicklung

Koroutinen und Threads

Michael Schöttner

Vorschau

Heinrich Heine
Universität Düsseldorf

- Motivation
- Quasi-Parallelität
- Koroutinen
- Threads
- Zusammenfassung

Motivation

Heinrich Heine
Universität Düsseldorf

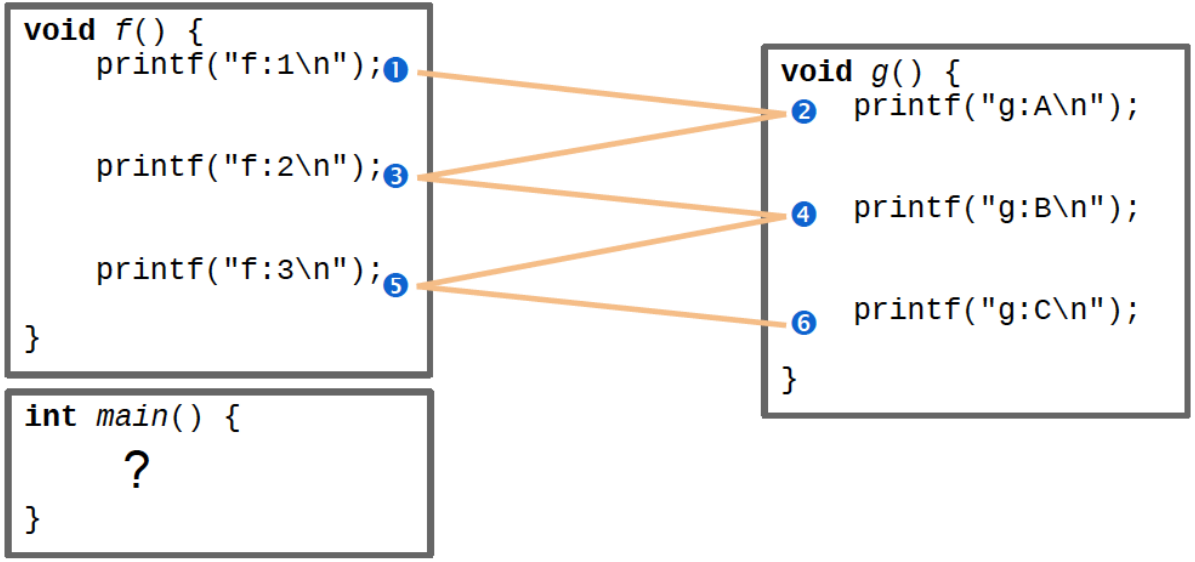
- Wir möchten Funktionen nebenläufig / asynchron ausführen
- Beispielsweise Daten übers Netzwerk empfangen oder von der Festplatte laden

Vorschau

- Motivation
- Quasi-Parallelität
- Koroutinen
- Threads
- Zusammenfassung

5.1 Quasi-Parallelität

- Gegeben: Funktionen f und g
- Ziel: f und g sollen „versetzt“ ablaufen



Quasi-Parallelität: Versuch 1

Heinrich Heine
Universität Düsseldorf

```
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();
}
```

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

Das funktioniert so natürlich nicht...

Quasi-Parallelität: Versuch 2

Heinrich Heine
Universität Düsseldorf

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

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

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

```
int main() {
  f();

}
```

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


    printf("g:B\n");


    printf("g:C\n");

}
```

So geht es wohl auch nicht...

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

Quasi-Parallelität: Versuch 3

Heinrich Heine
Universität Düsseldorf

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

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

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

```
int main() {
  f();

}
```

So schon
gar nicht!

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

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

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

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

Quasi-Parallelität: Versuch 4

Heinrich Heine
Universität Düsseldorf

```
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 (*volatile f)();
void (*volatile g)();

int main() {
    f = f_start;
    g = g_start;
    f();
}
```

Und so?

Klappt!

```
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);
}
```

```
os@ios>gcc-2.95 -o coroutine coroutine.c
os@ios>./coroutine
f:1
g:A
f:2
g:B
f:3
g:C
```

Erstes Fazit

Heinrich Heine
Universität Düsseldorf

- Quasi-Parallelität zwischen zwei Funktions-Ausführungen kann nicht durch Funktionsaufrufe erreicht werden
  - einfache Funktionsaufrufe (Versuche 1 und 2) → laufen immer komplett durch
  - rekursive Funktionsaufrufe (Versuch 3) → dito, deshalb Endlosrekursion und Stacküberlauf

- Wir brauchen Funktionen, die „während der Ausführung“ verlassen und wieder betreten werden können
  - also ungefähr so wie in Versuch 4
    - Instruktions-Pointer der Ausführungen wird gespeichert, mit goto wieder aufgenommen
  - aber bitte ohne die damit einhergehenden Probleme
    - direkte Sprünge aus und in Funktionen sind in C undefiniert! (goto über Zeiger ist ein gcc-“Feature“)
    - Zustand besteht aus mehr als dem Instruktions-Pointer – Was ist mit Registern, Stapel?

Vorschau

Heinrich Heine
Universität Düsseldorf

- Motivation
- Quasi-Parallelität
- Koroutinen
- Threads
- Zusammenfassung

Begriffe

Heinrich Heine
Universität Düsseldorf

- **Koroutine (engl. coroutine)**: Funktion deren Ausführung unterbrochen und später fortgesetzt werden kann.
- Koroutinen haben einen eigenen Registersatz + Stack
- Im Prinzip wie ein eigenständiger Thread, der aber dem Scheduler nicht bekannt ist
- Das Konzept stammt aus dem Bereich der Programmiersprachen
  - z.B. Kotlin, Rust, Python, Simula-67, …

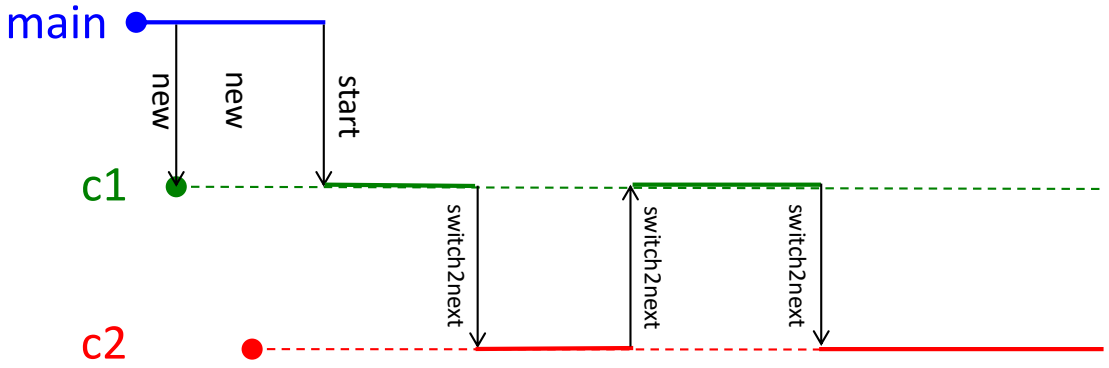
Sequentielle Ausführung von Koroutinen

Heinrich Heine
Universität Düsseldorf

- Koroutinen-Ausführungen bilden eine Fortsetzungsfolge
  - Zustand der Koroutine bleibt über Ein-/Austritte hinweg erhalten
- Alle Koroutinen sind gleichberechtigt
  - Ausführungsreihenfolge wird durch Koroutinen selbst bestimmt
    - Entweder statisch festgelegt (wie in hhuTOS)
    - Oder dynamisch bestimmt, durch die jeweils aktive Koroutine
  - Die Fortsetzungsfolge ist also programmiert, im Gegensatz zu einer dynamischen Ablaufplanung durch einen Scheduler

Ausführung von Koroutinen in hhuTOS

- Wir programmieren einen festen Ablaufplan für die Koroutinen, indem wir diese vor ihrer Erzeugung zyklisch verketten.
- Jede Koroutine ruft dann einfach `switch2next` auf und schaltet immer auf die nachfolgende Koroutine um



- Wir implementieren Koroutinen als Ergänzung bzw. als Vorstufe für unsere kooperativen und preemptiven Threads

Vorschau

Heinrich Heine
Universität Düsseldorf

- Motivation
- Quasi-Parallelität
- Koroutinen
- Threads
- Zusammenfassung

Threads

Heinrich Heine
Universität Düsseldorf

- „Ein Thread ist eine Koroutine die dem Scheduler bekannt ist“
- Scheduler verwaltet Threads und entscheidet, wann welcher Thread rechnen darf (siehe nächstes Kapitel)
- Ein Thread gehört immer zu einem Prozess(-adressraum)
  - In hhuTOS haben wir nur einen Prozess und damit nur einen Adressraum
- Ursprung der Konzepte:
  - Multi-Threading ist historisch (eher) ein Betriebssystemmerkmal
  - Koroutinen-Unterstützung ist historisch (eher) ein Sprachmerkmal

Lightweight Threads / Fibers

Heinrich Heine
Universität Düsseldorf

- Einige Programmiersprachen unterstützen Koroutinen durch den Compiler, kombiniert mit Laufzeitfunktionen, um asynchrone Programmierung zu unterstützen

- Primär für blockierenden I/O genutzt
  - Wenn ein Aufruf blockiert wird implizit die CPU abgegeben
  - Ein Executor schaltet dann auf die nächste Koroutine um
  - Hier sind die Koroutinen nicht fest nacheinander verdrahtet

- Rust: async & await
- Java: Project Loom

Vorschau

- Motivation
- Quasi-Parallelität
- Koroutinen
- Threads
- Zusammenfassung

Zusammenfassung

Heinrich Heine
Universität Düsseldorf

- Koroutinen sind Funktion deren Ausführung unterbrochen werden kann.
  - Im Prinzip wie ein Thread, der jedoch dem Betriebssystem nicht bekannt ist
- Threads sind Koroutinen die durch den Scheduler im Betriebssystem verwaltet werden
- Wir implementieren Koroutinen als Ergänzung bzw. als Vorstufe für unsere kooperativen und preemptiven Threads