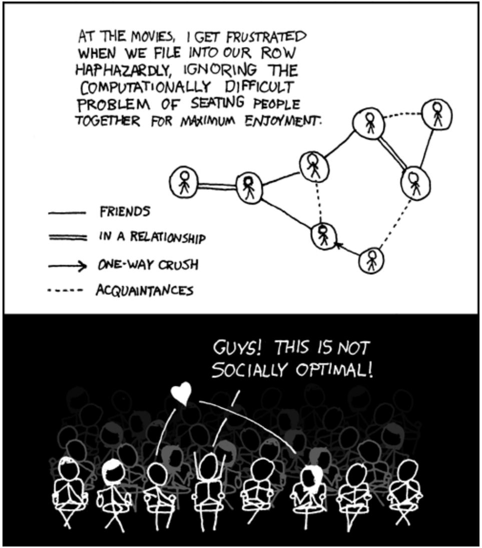
D&A - Übungsstunde 13

Diese Folien basieren auf denjenigen der Vorlesung, wurden aber durch den Assistenten Adel Gavranović adaptiert und erweitert

Comic der Woche



xkcd

Übersicht

Heutiges Programm

Intro
Follow-up
[code]expert
Threads
Quiz (Ausgaben)
Amdahl & Gustavson
Performanzmodell
Quiz (Scheduling)
In-Class Code-Example
Wiederholung Theorie
Alte Prüfungsfragen
Nächste Übung

n.ethz.ch/~agavranovic

- Link zum Material für die Übungsstunden
- Webseite des Assistenten
- Mail an Assistenten

1. Intro

Intro

- Viel zu tun; let’s go!

2. Follow-up

Follow-up aus vorherigen Übungsstunden

- –

3. [code]expert

Allgemeines zu [code]expert

- –

Fragen zu [code]expert eurerseits?

$\longrightarrow$ Mail

4. Threads

Parallele Programmierung

Parallele Programmierung = parallele Durchführung mehrerer Berechnungen

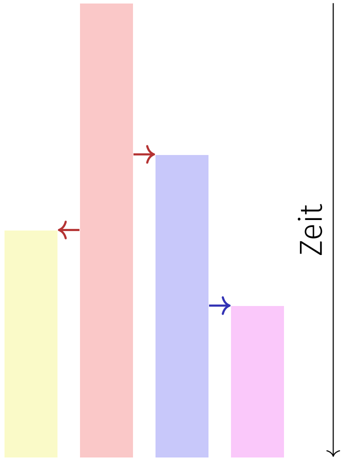
Etwas Terminologie

- **Tasks** sind Berechnungen, die durchgeführt werden müssen. Unabhängige Berechnungen können parallel ausgeführt werden.
- **Threads** sind parallele Ausführungen, die Aufgaben ausführen.
- Gemeinsam genutzte Ressourcen: was für die Ausführung von Aufgaben benötigt wird, aber gemeinsam genutzt werden muss, da es nicht für jede Aufgabe eine eigene Ressource gibt.

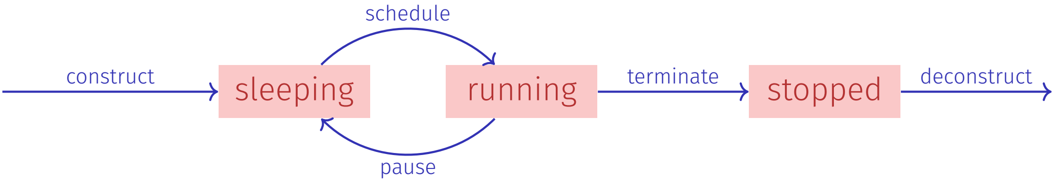
Threads abspalten (forking)

Das Abspalten (Forking) eines Threads bedeutet den Start einer neuen, parallelen Berechnung

- Haupt-Thread forked (erzeugt) einen neuen Thread
- Das Forking erfolgt durch die Erstellung eines neuen Thread-Objekts
  `std::thread(func, args...)`
- Der Haupt-Thread ist der Eltern-Thread seines Kind-Threads
- Jeder Thread kann weitere Threads abspalten



Thread-Lebenszyklus (vereinfacht)



Der **Scheduler** des Betriebssystems entscheidet

- welcher Thread als nächstes ausgeführt werden kann (schedule)
- auf welchem Kern er ausgeführt werden soll
- wann er wieder pausieren/schlafen soll

Das Umschalten zwischen schlafenden und laufenden Threads wird als **Kontextwechsel** bezeichnet

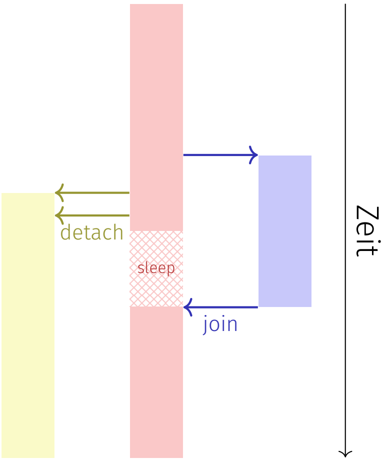
Join und Detach

`other_thread.join()` bedeutet warten, bis **`other`** fertig ist.

- Der aktuelle Thread schläft, bis der **`other`** beendet ist (wenn er das bereits getan hat, ist kein Schlafen notwendig)

`other_thread.detach()` bedeutet, dass kein anderer Thread auf die Beendigung von **`other`** warten wird

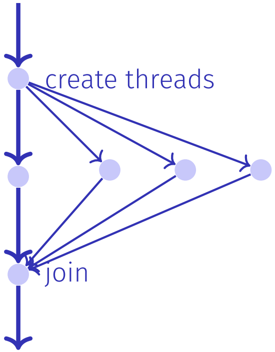
- Nützlich für nicht terminierende Prozesse (z. B. Server) und reaktive Systeme (z. B. GUIs)
- Beendet sich spätestens zusammen mit dem Haupt-Thread **`int main()`**



C++11 Threads

```cpp
void hello(int id){
  std::cout << "hello from " << id << "\n";
}

int main(){
  std::vector<std::thread> tv(3);
  int id = 0;
  for (auto & t:tv)
    t = std::thread(hello, ++id);
  std::cout << "hello from main \n";
  for (auto & t:tv)
    t.join();
  return 0;
}
```



Nichtdeterministische Ausführung!

Eine Ausführung:

```
hello from main
hello from 2
hello from 1
hello from 0
```

Andere Ausführung:

```
hello from 1
hello from main
hello from 0
hello from 2
```

Andere Ausführung:

```
hello from main
hello from 0
hello from hello from 1
2
```

Technische Details I

- Beim Erstellen von Threads werden auch Referenzparameter kopiert, ausser man gibt explizit std::ref bei der Konstruktion an.

```
void calc( std::vector<int>& very_long_vector ){
  // doing funky stuff with very_long_vector
}
int main(){
  std::vector<int> v( 1000000000 );
  std::thread t1( calc, v );          // bad idea, v is copied
  // here v is unchanged
  std::thread t2( calc, std::ref(v) ); // good idea, v is not copied
  // here v is modified
  std::thread t2( [&v]{calc(v)}; } ); // also good idea
  // here v is modified
  // ...
```

Technische Details II

- Threads können nicht kopiert werden.

```
{
  std::thread t1(hello);
  std::thread t2;
  t2 = t1; // compiler error
  t1.join();
}
{
  std::thread t1(hello);
  std::thread t2;
  t2 = std::move(t1); // ok
  t2.join();
}
```

5. Quiz (Ausgaben)

Quiz

```
void print(char c); // output character c

void A(char value){
  if (value != 'D'){
    std::thread t(A,value+1);
    print(value);
    t.join();
  }
}

int main(){
  std::thread t(A,'A');
  t.join();
}
```

mögliche Ausgabe(n)?

ABC
ACB
BAC
BCA
CAB
CBA

6. Amdahl & Gustavson

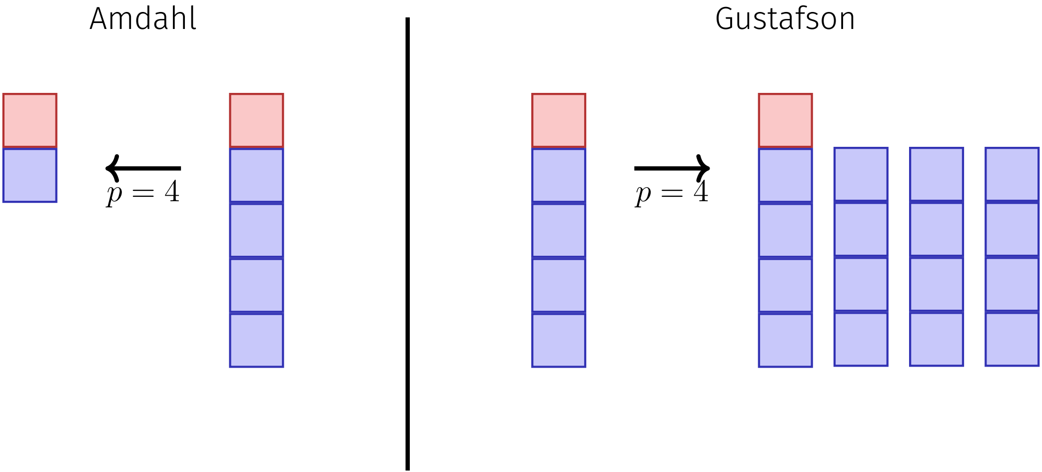
Speedup, Performanz und Effizienz

Gegeben

- fixierte Rechenarbeit W (Anzahl Rechenschritte)
- Sequentielle Ausführungszeit sei T_1
- Parallele Ausführungszeit T_p auf p CPUs

	Ausführungszeit	Speedup	Effizienz
Perfektion (linear)	$T_p = T_1/p$	$S_p = p$	$E_p = 1$
Verlust (sublinear)	$T_p > T_1/p$	$S_p < p$	$E_p < 1$
Hexerei (superlinear)	$T_p < T_1/p$	$S_p > p$	$E_p > 1$

Amdahl vs. Gustafson



Amdahl vs. Gustafson, oder wen juckt's?

Amdahl	**Gustafson**
Pessimist	Optimist
starke Skalierung	schwache Skalierung

$\Rightarrow$ Methoden müssen entwickelt werden so dass sie einen möglichst kleinen sequenziellen Anteil haben.

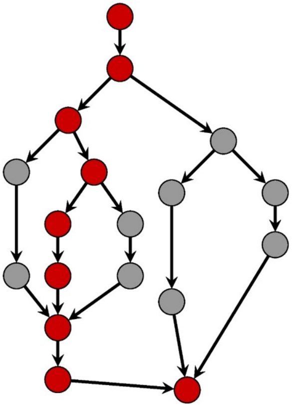
7. Performanzmodell

Performanzmodell

- T_1: **Arbeit:** Zeit für die gesamte Berechnung auf einem Prozessor
- T_p: Ausführungszeit auf p Prozessoren
- T_∞: **Zeitspanne**: Kritischer Pfad. Ausführungszeit auf ∞ Prozessoren. Längster Pfad von der Wurzel zur Senke.
- T_1/T_∞: **Parallelität:** breiter ist besser
- Untere Grenzen

$T_p \geq T_1/p$ Arbeitsgesetz

$T_p \geq T_\infty$ Zeitspannengesetz



Greedy Scheduler

Greedy Scheduler: teilt zu jeder Zeit so viele Tasks zu Prozessoren zu wie möglich.

Theorem 1

Auf einem idealen Parallelrechner mit p Prozessoren führt ein Greedy-Scheduler eine mehrfädige Berechnung mit Arbeit T_1 und Zeitspanne T_∞ in Zeit

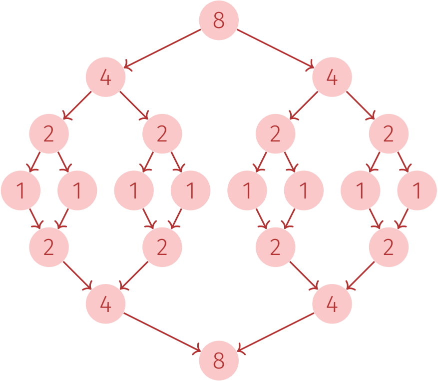
$$T_p \leq T_1/p + T_\infty$$

aus.

8. Quiz (Scheduling)

Quiz: Scheduling

Die folgende Abbildung zeigt einen Task-Graphen eines Algorithmus. Die Zahl an den Knoten bezeichnet die Ausführungszeit für den jeweiligen Berechnungsschritt.



$$T_\infty = 29$$

$$T_1 = 56$$

$$T_4 \leq 56/4 + 29 = 43$$

$$T_8 \leq 56/8 + 29 = 36$$

9. In-Class Code-Example

Ein mysteriöser Effekt $\longrightarrow$ [code]expert

10. Wiederholung Theorie

Race Conditions (Wettlaufsituationen)

Data Race Fehlerhaftes Programmverhalten verursacht durch ungenügend synchronisierten Zugriff zu einer gemeinsam genutzten Resource, z.B. gleichzeitiges Lesen/Schreiben oder Schreiben/Schreiben zum gleichen Speicherbereich.

Bad Interleaving Fehlerhaftes Programmverhalten verursacht durch eine unglückliche Ausführungsreihenfolge eines Algorithmus mit mehreren Threads, selbst dann wenn die gemeinsam genutzten Resourcen anderweitig gut synchronisiert sind.

Speichermodelle

Wann und ob Effekte von Speicheroperationen für Threads sichtbar werden, hängt also von Hardware, Laufzeitsystem und der Programmiersprache ab.

Ein **Speichermodell** (z.B. das von C++) gibt Minimalgarantien für den Effekt von Speicheroperationen.

- Lässt Möglichkeiten zur Optimierung offen
- Enthält Anleitungen zum Schreiben Thread-sicherer Programme

C++ gibt zum Beispiel **Garantien, wenn Synchronisation mit einer Mutex verwendet** wird.

Counter Problem

```
std::vector<std::thread> tv(10);
int counter {0};
for (auto & t:tv)
  t = std::thread([&]{
    for (int i =0; i<100000; ++i){counter++;} // race!!
  });
for (auto & t:tv)
  t.join();
std::cout << "count= "<< counter << std::endl;
```

Counter Lösung 1

```
std::vector<std::thread> tv(10);
std::mutex lock;
int counter {0};
for (auto & t:tv)
  t = std::thread([&]{
  for (int i =0; i<100000; ++i){
    mutex.lock(); counter++; mutex.unlock(); // synchronized!
  }});
for (auto & t:tv)
  t.join();
std::cout << "count= "<< counter << std::endl;
```

Counter Lösung 2

```
std::vector<std::thread> tv(10);
std::atomic<int> counter {0};
for (auto & t:tv)
  t = std::thread([&]{
    for (int i =0; i<100000; ++i){counter++;} // atomic!!
  });
for (auto & t:tv)
  t.join();
std::cout << "count= "<< counter << std::endl;
```

Quiz: Was ist hier falsch?

```
void exchangeSecret(Person & a, Person & b) {
  a.getMutex()->lock();
  b.getMutex()->lock();
  Secret s = a.getSecret();
  b.setSecret(s);
  a.getMutex()->unlock();
  b.getMutex()->unlock()
}
```

Deadlock (Verklemmung)

Thread 1:
`exchangeSecret(p1, p2);`

Thread 2:
`exchangeSecret(p2, p1);`

Was tun?

Mögliche Lösung

```
void exchangeSecret(Person & a, Person & b) {
  std::mutex* first;
  std::mutex* second;
  if (a.name < b.name){
    first = a.getMutex(); second = b.getMutex();
  } else {
    first = b.getMutex(); second = a.getMutex();
  }
  first->lock();
  second->lock();
  Secret s = a.getSecret();
  b.setSecret(s);
  first->unlock();
  second->unlock();
}
```

Deadlocks und Races

- Nicht einfach zu sehen
- Schwierig zu debuggen
- Treten u.U. selten auf
- Testing genügt nicht
- Eigentlich muss man die Korrektheit des Codes formal beweisen

Vorsicht und Sorgfalt ist gefragt beim Programmieren mit Locks!

Quiz

```
void print(char c); // output c
std::mutex m1, m2;
char value;

void B(){
  m1.lock(); m2.lock();
  print(value++);
  m2.unlock(); m1.unlock();
}
void A(){
  m2.lock(); m1.lock();
  print(value++);
  m1.unlock(); m2.unlock();
}
```

```
int main(){
  value = 'A';
  print(value++);
  std::thread t1(A);
  std::thread t2(B);
  t1.join();
  t2.join();
}
```

mögliche Ausgabe(n)?

AA
Und das Programm terminiert nicht!

Bedingungsvariabeln

Bedingungsvariabeln (*condition variables*) erlauben einem Thread effizient auf eine gewisse Bedingung zu warten.

Sobald sich die Bedingung geändert hat (oder geändert haben könnte), benachricht der verursachende Thread den/die wartenden Threads.

Bedingungsvariablen

```
class Buffer {
...
public:
    void put(int x){
        guard g(m);
        buf.push(x);
        cond.notify_one();
    }
    int get(){
        guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        int x = buf.front(); buf.pop();
        return x;
    }
};
```

Bedingungsvariablen

```
class Buffer {
...
public:
    void put(int x){
        guard g(m);
        cond.notify_one();
        buf.push(x);
    }
    int get(){
        guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        int x = buf.front(); buf.pop();
        return x;
    }
};
```

Ist das auch korrekt?

Antwort

- Es spielt hier keine Rolle, wo das Signalling passiert.
- Der Effekt des Signalling tritt erst auf, wenn der Thread die kritische Region verlässt.

11. Alte Prüfungsfragen

Alte Prüfungsfragen

Frage

Die Analyse eines Programms hat eine Beschleunigung von 2 gezeigt, wenn es auf 9 Prozessorkernen läuft. Wie groß ist der serielle Anteil nach Gustafsons Gesetz?

Antwort

Mit der Formel von Gustafsons Gesetz $Sp = p - f \cdot (p - 1)$, setzen wir die gegebenen Werte $Sp = 2$ und $p = 9$ ein, um $2 = 9 - f \cdot 8$ zu bekommen. Nach Umstellung erhalten wir $7 = f \cdot 8$. Wenn wir nach f (dem seriellen Anteil) auflösen, finden wir $f = \frac{7}{8} = 0.875$.

Alte Prüfungsfragen

Frage

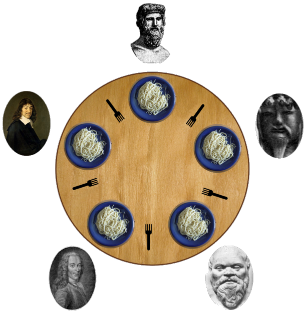
Sie führen eine Messung Ihres Programms mit einer sehr großen Anzahl von Prozessorkernen durch. Die Messungen deuten darauf hin, dass die Beschleunigung (mit beliebig vielen Prozessorkernen) von oben durch $S_\infty = 2.5$ begrenzt ist. Was ist die bestmögliche obere Grenze für die Beschleunigung mit 6 Kernen, vorausgesetzt, dass Amdahls Gesetz für Ihr Problem gilt?

Antwort

Mit der Formel von Amdahls Gesetz $Sp \leq \frac{1}{f+\frac{1-f}{p}}$ und $S_\infty = \frac{1}{f} = \frac{5}{2}$, finden wir $f = \frac{2}{5}$. Setzen wir f und $p = 6$ in Amdahls Gesetz ein, um $S_6 \leq \frac{1}{\frac{0.4}{1}+\frac{0.6}{6}}$.
Daher ist $S_6 \leq 2$.

12. Nächste Übung

Dining Philosophers



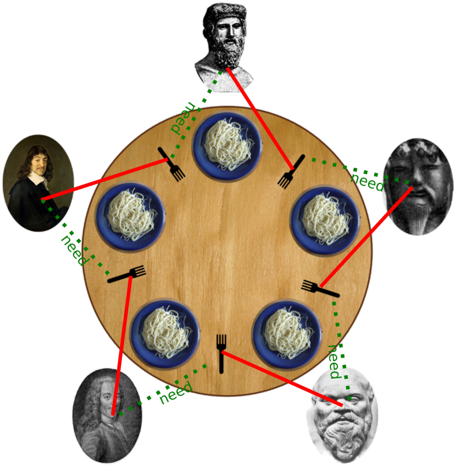
- Philosophen denken und essen abwechslungsweise. Zum Essen brauchen sie zwei Gabeln.
- Philosoph = Thread, Gabel = Lock.

Dining Philosophers - Pseudocode

```
while(true) {
    think();
    acquire_fork_on_left_side();
    acquire_fork_on_right_side();
    eat();
    release_fork_on_right_side();
    release_fork_on_left_side();
}
```

- Problem mit diesem Program?

Dining Philosophers - Deadlock



- Lösung?

Dining Philosophers

- Zyklische Abhängigkeit brechen
- Beispielsweise: Philosoph fünf nimmt erste **rechte** Gabel.
- Allgemeine Möglichkeit: Lock Ordnung definieren. Dann immer in dieser Reihenfolge locken.