Programmierparadigmen

Kapitel 4c
Parallele Programmierung in C++

(Diese Folien beruhen auf einem Foliensatz von Prof. Dr. Kai Uwe Sattler)

Überblick

- Grundlagen
- Parallele Programmierung in Erlang
- Parallele Programmierung in C++
  - Kurzer Nachtrag zu C++: `auto` und Lambda-Expressions
  - Threads
  - Datenparallele Verarbeitung
  - Kommunikation zwischen Threads
  - Task-Parallelität
- Parallele Programmierung in Java

Typinferenz durch C++-Compiler mit `auto` (1)

- Typinferenz bezeichnet die automatische Bestimmung des Typs eines Ausdrucks in einer Programmiersprache
- Ab C++ Version 11 kann das Schlüsselwort `auto` bei Deklaration von:
  - Variablen und
  - Funktionen

  den C++-Compiler anweisen, den Typ der zu deklarierenden Variable bzw. Rückgabewert der Funktion selbst zu berechnen
- Auf der folgenden Folie wird der Operator `typeid` verwendet, um den Typ der Variablen zur Laufzeit zu bestimmen
  - Hier wird er zur Bestimmung statischer Typen verwendet, aber `typeid` unterstützt auch dynamische Typinferenz für Objekte von Klassenhierarchien („Run-Time-Type-Inference“, RTTI)

Typinferenz durch C++-Compiler mit `auto` (2)

```
1  int main() {
2    auto x = 1;
3    auto y = 2.3;
4    auto z = 4.56f;
5    auto e = 'A';
6    auto p = &x; // pointer to int
7    auto q = &p; // pointer to a pointer
8
9    std::cout << typeid(x).name() << “, “ <<
10                typeid(y).name() << “, “ <<
11                typeid(z).name() << “, “ <<
12                typeid(e).name() << “, “ <<
13                typeid(p).name() << “, “ <<
14                typeid(q).name() << std::endl;
15 }
```

- Ausgabe: `i, d, f, c, Pi, PPi`

Typinferenz durch C++-Compiler mit `auto` (3)

- Achtung: `auto` wird zu `int` in Fällen, in denen eine Integer-Referenz zugewiesen wird (entsprechend bei anderen Basisdatentypen)

```
int&  fun() {static int i = 1; return i;}
// ...
auto  g = fun();
auto& h = fun();
```

- Hier ergibt sich:
  - Typ der Variable g: `int`
  - Typ der Variable h: `int&`
- Um in solchen Fällen einen Referenz-Typ zu erhalten, muss also `auto&` verwendet werden.
- Die Ermittlung des tatsächlichen Typs erfolgt bei `auto` immer zur Compile-Zeit!

Typinferenz durch C++-Compiler mit `auto` (4)

- Vor- und Nachteile:
  - Erspart Denkarbeit bei der (initialen) Programmerstellung:
    - *„Warum den Menschen mit Dingen beschäftigen, die der Compiler ohnehin besser weiß?“*
    - *„Ich habe jetzt keine Lust darüber nachzudenken, was ich hier für einen Typ brauche, überlasse das lieber dem Compiler und hoffe, dass alles gut geht…“*
    - Erfordert dafür allerdings ggf. zusätzliche Denkarbeit beim späteren Lesen/Anpassen/Fehlerbereinigen des Codes :-(
    - Daher bitte sehr sorgfältig mit diesem Konstrukt umgehen
  - Kann jedoch in manchen Situationen hilfreich/sinnvoll sein:
    - Hilft bei komplexen Iteratoren, zu lange Deklarationen z.B. in For-Schleifen zu vermeiden
    - Ab C++ Version 14 wird auch die Deklaration von Parametertypen und Rückgabetyp für Lambda-Expressions (siehe nächste Folie) unterstützt

Lambda-Expressions (1)

- In vielen modernen Programmiersprachen ermöglichen es Lambda-Expressions, anonyme Funktionen zu realisieren.
- In C++ können anonyme Funktionen folgendermaßen definiert werden:
  - `[Capture]<Template>(Parameter) -> Type { Body }`
    - `Capture`: Übertrag der angegebenen Symbole in den Gültigkeitsbereich des Lambda-Ausdrucks (→ realisiert „Closure“)
    - `Template`: Liste der Templateparameter (optional)
    - `Parameter`: Liste der Übergabeparameter (optional)
    - `Type`: Rückgabetyp (optional)
    - `Body`: Funktionsrumpf
  - Die eckigen Klammern `[ ]` sind hier nicht als „optionale Angabe“ zu verstehen, sondern müssen explizit hingeschrieben werden

Lambda-Expressions (2)

- Lambda-Expressions können an Funktionen übergeben oder auch Variablen zugewiesen werden:

```
template<typename T>
void sort(std::vector<T>& a, std::function<bool(T, T)> F) {…};

std::vector<int> v = {5, 17, 3};
auto smaller = [](auto a, auto b){return a < b;};

// ...
sort<int>(v, smaller);
// ...
sort<int>(v, [](auto a, auto b){return a > b;}); // bigger
```


Lambda-Expressions (3)

- **Closures** sind ursprünglich ein Konzept funktionaler Programmier-sprachen, das mittlerweile auch in vielen imperativen Sprachen unterstützt wird und das es erlaubt, nicht-sichtbare aber kontrolliert veränderbare Bereiche zu erstellen.
- In C++ werden Closures im Kontext von Lambda-Expressions realisiert
- Hierzu werden die Variablen des umgebenden Sichtbarkeitsbereichs, auf die im Body zugegriffen wird, bei `Capture` explizit (mit ihrem Namen) oder implizit (mittels „=“ ~ als Kopie oder „&“ ~ per Referenz) angegeben

```
int a = 0, b = 1;
auto l1 = [a](){return ++a == 1;}; // a danach unverändert
auto l2 = [&a](){return ++a == 1;};// a danach verändert
auto l3 = [=](){return ++a == b;}; // a danach unverändert
auto l4 = [&](){return ++a == b;}; // a danach verändert
```


Threads in C++ (1)

- **Thread** (Englisch für „Faden“) := Leichtgewichtige Ausführungseinheit (Folge von Anweisungen) innerhalb eines sich in Ausführung befindlichen Programms
  - Threads führen nach ihrem Start eine (initiale) Funktion aus
  - Threads teilen sich den Adressraum ihres Prozesses
  - Dies ermöglicht den Zugriff auf gemeinsam verwendete Speicherbereiche, wodurch ein Bedarf für Koordination solcher Zugriffe besteht
  - In C++: Threads sind Instanzen der Klasse `std::thread`

Threads in C++ (2)

```
#include <thread>
#include <iostream>

void say_hello() {
    std::cout << "Hello Concurrent C++\n";
}

int main() {
    std::thread t(say_hello);
    t.join();
}
```

- Das Starten des Threads ist hier nicht sinnvoll eingesetzt; das Beispiel dient nur zur Demonstration der Syntax

Alternative Erzeugung von Threads

- Über Lambda-Expression

```
std::thread t([]() {do_something(3); }); // ~ Lazy Eval.
```

- Mit Instanz einer Klasse – erfordert Überladen von `operator()`

```
struct my_task {
    void operator()() const { do_something(); }
};

my_task tsk;
std::thread t1(tsk);          // mit Objekt
std::thread t2{ my_task() };  // mit temporärem Objekt
```


Parameter-Übergabe bei Thread-Erzeugung

- Kann über zusätzliche Argumente des Thread-Konstruktors erfolgen
- Vorsicht bei Übergabe von Referenzen:
  - Eltern-Thread könnte ggf. schon vor dem erzeugten Thread beendet (und dabei die entsprechende Variable freigegeben) werden
  - Threads können selber wiederum Threads erzeugen…
  - Der Thread-Konstruktor unterstützt variable Anzahl von Parametern

```
void fun(int n, const std::string& s) {
    for (auto i = 0; i < n; i++)
        std::cout << s << " ";
    std::cout << std::endl;
}

std::thread t(fun, 2, "Hello");
t.join();
```


Warten auf Threads (1)

- Die Funktion `t.join()` wartet auf Beendigung des Threads `t`
- Der aktuelle Thread wird blockiert bis Thread `t` beendet ist
- Freigabe der Ressourcen des Threads
- Ohne `join()` gibt es keine Garantie, dass der Thread `t` beendet wird bevor die Ressourcen des std::thread-Objekts freigegeben werden

```
std::thread t([]() {do_something(3); });
t.join();
```

```
{ std::thread t([]() {do_something(3); }); }
// hier würde ein Speicherfehler auftreten
```


Warten auf Threads (2)

- Erscheint die Ausgabe?

```
#include <iostream>
#include <thread>
#include <chrono>

int main() {
    std::thread t([]() {
        std::this_thread::sleep_for(
            std::chrono::seconds(1));
        std::cout << "Hello" << std::endl;
    });
}
```


Hintergrund-Threads

- Threads können auch im Hintergrund laufen, ohne dass auf ihr Ende gewartet werden muss
- „Abkoppeln“ durch `detach()`
  - Der Thread läuft danach unter Kontrolle des C++-Laufzeitsystems
  - Danach ist `join()` nicht mehr möglich
- Soll ein Thread abgekoppelt ablaufen können, ist darauf zu achten, dass er nicht auf Variablen zugreift, die nach seiner Erzeugung durch Beenden seines „Eltern-Threads“ gelöscht werden könnten – daher:
  - Keine Variablen als Referenzen übergeben
  - In diesem Fall sorgt das detach() auch dafür, dass der Thread nicht mehr zum Abschluss auf Variablen des (evtl. schon gelöschten) Thread-Objekts zugreift

Thread Identifikation

- Thread-Identifikator vom Typ `std:thread::id`
- Ermittlung über Methode `get_id()`

```
void fun() {
    std::cout << "Hello from "
              << std::this_thread::get_id()
              << std::endl;
}

std::thread t(fun);
t.join();
```


Berechnung von Fibonacci-Zahlen in C++

- Hier: nicht-rekursive Variante

```
unsigned int fibonacci(unsigned int n) {
    if (n == 0)
        return 0;

    unsigned int f0 = 0, f1 = 1, f2;
    for (unsigned int i = 1; i < n; i++) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
    }
    return f1;
}
```


Parallele Berechnung mehrerer Fibonacci-Zahlen (1)

- Einfache Lösung (ähnlich zu Erlang): pro Zahl ein Thread

```
1  std::vector<std::thread> threads;
2  unsigned int inputs[20];
3  unsigned int results[20];
4
5  for(unsigned int i = 0; i < 20; i++) {
6     unsigned int f = rand() % 30; inputs[i] = f;
7     threads.push_back(std::thread([&, i, f]() {
8                                    results[i] = fibonacci(f);
9                                   }));
10 } // of <for i>
11
12 for(std::thread& t : threads) t.join();
```


Parallele Berechnung mehrerer Fibonacci-Zahlen (2)

- Zeile 1: Vector von Threads anlegen (Länge variabel)
- Zeile 2-3: Felder für Eingabe- und Ergebniswerte anlegen
- Zeile 6: Zufallszahl zwischen 0 und 30 erzeugen und merken
- Zeilen 7-9: Thread zur Berechnung der Fibonacci-Zahl erzeugen und Ergebnis im Feld `results` speichern
- Zeile 12: Warten auf Beendigung der Threads

- Aber:
  - Zugriff auf gemeinsame Ressource (Ergebnisfeld `results`)!
  - Anzahl der erzeugten Threads = Anzahl Fibonacci-Zahlen
  - Kann wie auch schon bei Erlang zu schlechterer Performance als für die sequentielle Variante führen

Kontrolle der Anzahl der Threads

- Die Erzeugung von und das Umschalten zwischen Threads ist mit Kosten verbunden, bspw.:
  - Speicherplatz für Zwischenspeicherung von Registerinhalten
  - CPU-Zeit für Speichern und Laden von Registern bei Umschalten
- Jedes System unterstützt nur eine begrenzte Anzahl von Hardware-Threads (bedingt durch Anzahl Prozessoren/Cores, Unterstützung Hyperthreading Ja/Nein)
- Es sollten idealerweise nicht mehr Threads zu einem Zeitpunkt erzeugt werden als Hardware-Threads auf dem System unterstützt werden
  - Ermittlung mittels `std::thread::hardware_concurrency()`
  - Es bleibt das Problem, bei Programmentwicklung zu entscheiden, wieviele Threads zu einem gegebenen Zeitpunkt ideal wären
  - Programmierabstraktionen sinnvoll: Threadpools, Task-Libraries, ...

Unterstützung durch APIs & Frameworks

- Parallele Programmierung kann durch spezielle Konstrukte einer Programmiersprache, Application Programming Interfaces (APIs) oder Frameworks unterstützt werden
- OpenMP:

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {...}
```

- Parallele Algorithmen in C++17:

```
std::for_each(std::execution::par_unseq,
vec.begin(), vec.end(), [](auto& item) { ... });
```

- Intel TBB (Threading Building Blocks):

```
tbb::parallel_for(tbb::blocked_range<int>(0,vec.size()),
                  [&](tbb::blocked_range<int> r) { ... }
```


Probleme nebenläufiger Ausführung (1)

```
1  struct jawsmith {
2     std::string msg;
3
4     jawsmith(const std::string& m) : msg(m) {}
5
6     void operator()() const {
7        for(;;) {
8           std::this_thread::sleep_for(
9              std::chrono::seconds(1));
10          for (auto& c : msg) {
11             std::cout << c << std::flush;
12          } // of <for c>
13       } // of <for ;;>
14    } // of operator()
15 };
```


Probleme nebenläufiger Ausführung (2)

```
std::thread t1 { jawsmith("DASISTEINELANGENACHRICHT") };
std::thread t2 { jawsmith("dieistaberauchnichtkurz") };
```

- Ausgabe:

```
dDieistaberauchnichtkASISTEINELANGENACHurzRICHT...
```

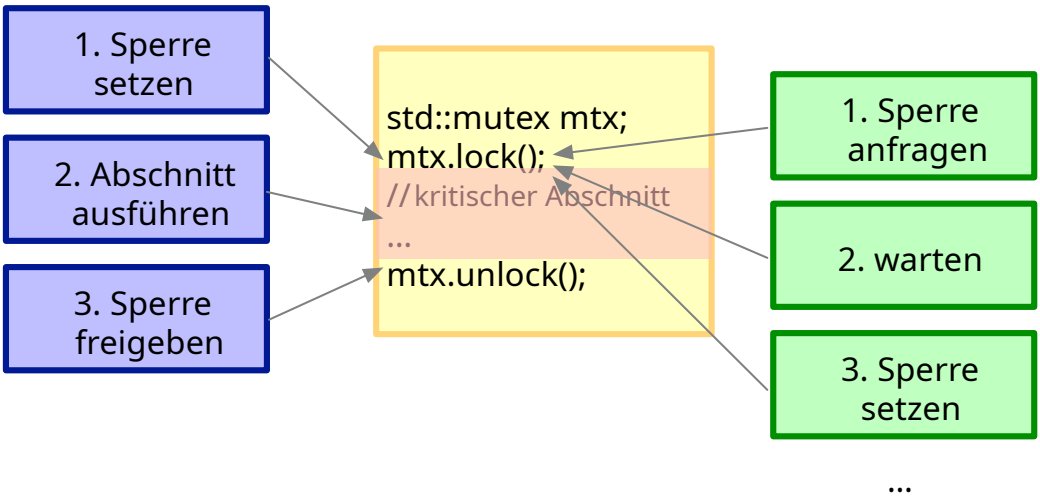
- Race Conditions („Wettlaufsituation“):
  - Das Ergebnis einer nebenläufigen Ausführung auf gemeinsamen Zustand (hier: Ausgabekanal) hängt vom zeitlichen Verhalten der Einzeloperationen ab
  - Erfordert in der Regel eine Koordination der nebenläufigen Ausführungen oder sofern möglich ihre Entkopplung (meist performanter)

Koordination durch wechselseitigen Ausschluss

- Der Zugriff auf den Ausgabekanal im Beispiel ist ein Beispiel für einen sogenannten „kritischen Abschnitt“
- **Kritischer Abschnitt** (Englisch: „Mutual Exclusion = mutex“):
  - Programmabschnitt in einem Thread, in dem auf eine gemeinsame Ressource (Speicher, Ausgabekanal, etc.) zugegriffen wird und der nicht parallel (oder zeitlich verzahnt) zu einem anderen Thread ausgeführt werden darf
- **Wechselseitiger Ausschluss**:
  - Es muss dafür gesorgt werden, dass die Ausführung kritischer Abschnitte in Threads nur im wechselseitigen Ausschluss erfolgt
  - Es darf immer nur ein Thread im kritischen Abschnitt sein

Mutex in C++

- Instanz der Klasse std::mutex
- Methoden zum Sperren (lock) und Freigeben (unlock)



Mutex-Varianten

- `mutex`: Standard-Mutex für exklusiven Zugriff
- `timed_mutex`: Mutex mit Timeout für Warten
  - `try_lock_for()`: versucht eine übergebene Zeit lang, den Mutex zu erlangen und gibt true/false zurück, je nachdem ob der Mutex erhalten wurde
- `recursive_mutex`: rekursives Mutex – erlaubt mehrfaches Sperren durch einen Thread, z.B. für rekursive Aufrufe
- `recursive_timed_mutex`: rekursives Mutex mit Timeout
- `shared_mutex`: Mutex, das gemeinsamen Zugriff (`lock_shared()`) mehrerer Threads oder exklusiven Zugriff (`lock()`) ermöglicht
- `shared_timed_mutex`: Mutex mit Timeout und gemeinsamen Zugriff

Lock Guards

- Wie Risiko verringern, dass ein Mutex `m` nicht wieder freigegeben wird?
- Konzept „RAII“ („Ressourcenbelegung ist Initialisierung’”)
  - Konstruktor ruft `m.lock()` und Destruktor ruft `m.unlock()` auf

```
std::vector<int> data;
std::mutex my_mtx;

void add(int val) {
    std::lock_guard<std::mutex> guard(my_mtx);
    data.push_back(val);
} // when variable guard disappears, the lock my_mtx is released

int get() {
    std::lock_guard<std::mutex> guard(my_mtx);
    return data.front();
 }
```


Lock Guards und Locks

- Beim Setzen von mehreren Sperren muss verhindert werden, dass es zu sogenannten „Deadlocks“ kommen kann
  - Beispiel: Zwei Threads versuchen gleichzeitig jeweils zwei gleiche Locks in unterschiedlicher Reihenfolge zu erhalten
- `std::unique_lock` erweiterte Variante von `std::lock_guard`, vermeidet aber sofortiges Sperren
- `std::lock` erlaubt gleichzeitiges deadlock-freies Sperren von zwei Mutexen
- Beispiele für Sperrstrategien:
  - `std::try_to_lock` versucht Sperre ohne Blockierung zu setzen
  - `std::adopt_lock` versucht nicht, ein zweites Mal zu sperren, wenn bereits durch den aktuellen Thread gesperrt
  - `std::defer_lock` führt keine Sperrung durch

Atomare Datentypen

- `std::atomic_flag` = sperrfreier, atomarer Datentyp
  - `clear()` setzt den Wert auf `false`
  - `test_and_set()` setzt den Wert atomar auf `true` und liefert den vorherigen Wert
- `std::atomic<bool>` = Variante, erlaubt Abfrage ohne Setzen
  - `operator=` atomare Wertzuweisung
  - `load()` liefert den aktuellen Wert
  - `exchange()` entspricht `test_and_set()`
- `std::atomic<T>` = generische Variante für weitere Datentypen
  - Damit keine inkonsistente Werte ausgelesen werden können, falls mehrere Bytes geschrieben werden müssen

Synchronisation über atomare Variable (1)

```
1   std::list<std::string> shared_space;
2   std::atomic<bool> ready = false;
3
4   void consume() {
5       while (!ready.load())
6           std::this_thread::sleep_for(
7                   std::chrono::milliseconds(10));
8       std::cout << shared_space.front() << std::endl;
9       shared_space.pop_front();
10  }
11
12  void produce() {
13      shared_space.push_back("Hallo!");
14      ready = true;
15  }
16
17  std::thread t1(consumer);
18  std::thread t2(producer);
19  ...
```

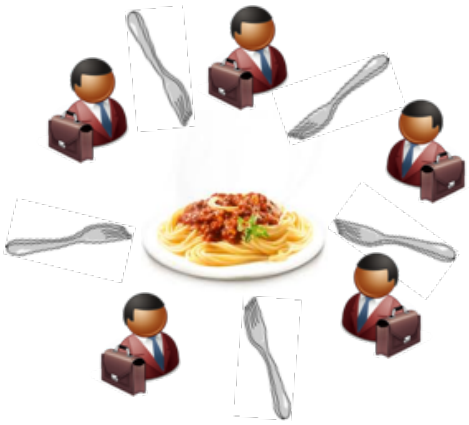

Synchronisation über atomare Variable (2)

- Z. 1: gemeinsam genutzte Liste, erfordert synchronisierten Zugriff
- Z. 2: atomare Boolesche Variable ready
- Z. 4/12: Konsument/Produzent-Threads
- Z. 5: atomares Prüfen der ready-Variablen
- Z. 6-7: kurz warten und neu versuchen
- Z. 8-9/13: Zugriff auf gemeinsame Liste
- Z. 14: atomares Setzen der Variablen ready

Task-Parallelität: Die 5 speisenden Philosophen

- Fünf Philosophen teilen sich eine Schüssel Spaghetti
- Fünf Gabeln, je eine zwischen zwei Philosophen
- Ein Philosoph kann nur mit zwei (benachbarten) Gabeln essen
- Gabeln werden nur nach dem Essen zurückgelegt
- Ein Philosoph durchläuft Zyklus von Zuständen:

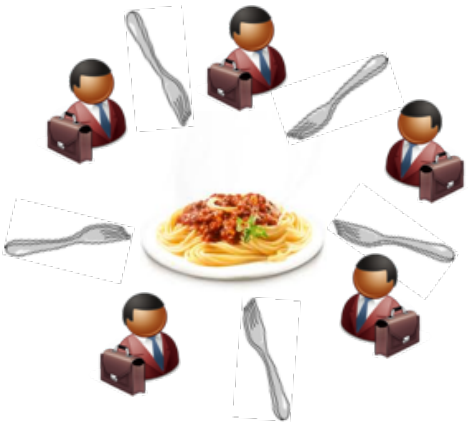
  denken →hungrig → essen → denken → …



Das Problem mit den Philosophen…

- Jeder greift die **linke** Gabel
- und wartet auf die **rechte** Gabel
- … und wartet …

Verklemmung!



Lösungsidee

- Immer beide Gabeln aufnehmen, d.h. wenn nur eine Gabel verfügbar ist: liegen lassen und warten
- Synchronisierter Zugriff auf Gabeln, d.h. in einem kritischen Abschnitt unter gegenseitigem Ausschluss
  - Erfordert „gleichzeitiges“ Erlangen zweier Mutexe
- Wecken von wartenden Philosophen

Verklemmungsfreies Sperren zweier Mutexe

- Lösung 1: Erlange beide Mutexe atomar und verwende dann RAII:

```
std::lock(mtx1, mtx2);
std::lock_guard<std::mutex> lk1(mtx1, std::adopt_lock);
std::lock_guard<std::mutex> lk2(mtx2, std::adopt_lock);
```

- Lösung 2: Bereite RAII vor und erlange dann beide Mutexe atomar:

```
std::unique_lock<std::mutex> lk1(mtx1, std::defer_lock);
std::unique_lock<std::mutex> lk2(mtx2, std::defer_lock);
std::lock(lk1, lk2);
```


Gabeln und Spaghetti-Teller

```
1  // Gabel = Mutex
2  struct fork {
3      std::mutex mtx;
4  };
5
6  // 1 Teller mit 5 Gabeln
7  struct spaghetti_plate {
8      // Benachrichtigung der Philosophen
9      std::atomic<bool> ready = false;
10     std::array<fork, 5> forks;
11 };
```


Die Philosophen-Klasse

```
1  class philosopher {
2      private:
3          int               id;
4          spaghetti_plate&  plate;
5          fork&             left_fork;
6          fork&             right_fork;
7
8      public:
9          philosopher(int n, spaghetti_plate& p) :
10             id(n), plate(p),
11             left_fork(p.forks[n]),           // 1. Gabel
12             right_fork(p.forks[(n + 1) % 5]) // 2. Gabel
13             {}
14     ...
15 };
```


Die Philosophen-Klasse: Hilfsmethoden

- Textausgabe erfordert synchronisierten Zugriff auf Ausgabekanal `cout` über globalen Mutex:

```
1  void say(const std::string& txt) {
2     std::lock_guard<std::mutex> lock(out_mtx);
3     std::cout << "Philosopher #" << id
4               << txt << std::endl;
5  }
```

- Hilfsmethode für zufällige Wartezeit in Millisekunden:

```
1  std::chrono::milliseconds wait() {
2     return std::chrono::milliseconds(rand() % 500 + 100);
3  }
```


Die Philosophen-Klasse: Essen

```
1  void eating() {
2     // Versuche, die Gabeln (verklemmungsfrei)
3     // aufzunehmen
4     std::lock(left_fork.mtx, right_fork.mtx);
5     std::lock_guard<std::mutex>
6        left_lock(left_fork.mtx, std::adopt_lock);
7     std::lock_guard<std::mutex>
8        right_lock(right_fork.mtx, std::adopt_lock);
9
10    // Essen simulieren
11    say(" started eating.");
12    std::this_thread::sleep_for(wait());
13    say(" finished eating.");
14 }
```


Die Philosophen-Klasse: Denken

```
1  void thinking() {
2      say(" is thinking.");
3      // Wenn Philosophen denken ...
4      std::this_thread::sleep_for(wait());
5  }
```


Das Leben eines Philosophen

- Zur Erinnerung: Ein überladener ()-Operator eines Objektes definiert die auszuführende Funktion eines Threads

```
1  void operator()() {
2      // Warten bis der Teller bereit ist
3      while (!plate.ready); // Warten und nichts tun
4
5      do {
6          // solange der Teller bereit ist
7          thinking();
8          eating();
9      } while (plate.ready);
10 }
```


Das Dinner: Initialisierung

```
1  // der Teller
2  spaghetti_plate plate;
3
4  // die 5 Philosophen
5  std::array<philosopher, 5> philosophers {{
6     { 0, plate }, { 1, plate },
7     { 2, plate }, { 3, plate },
8     { 4, plate }
9  }};
10
11 // Thread pro Philosoph erzeugen
12 std::array<std::thread, 5> threads;
13 for (int i = 0; i < threads.size(); i++) {
14    threads[i] = std::thread(philosophers[i]);
15 }
```


Das Dinner beginnt

- Beginn (und Ende) des Dinners über atomare Variable signalisieren
- Philosophen-Threads arbeiten ihre operator()-Methode ab

```
1  // das Essen beginnt und dauert 5 Sekunden ;-)
2  std::cout << "Starting dinner ..." << std::endl;
3  plate.ready = true;
4  std::this_thread::sleep_for(std::chrono::seconds(5));
5  // Abräumen
6  plate.ready = false;
7  // Warten auf Beendigung der Threads
8  for(std::thread& t : threads) t.join();
9  std::cout << "Dinner finished!" << std::endl;
```


Speisende Philosophen: Zusammenfassung

- „Philosophenproblem“: Klassisches Beispiel der Informatik zur Demonstration von Nebenläufigkeit und Verklemmung
- Das Problem wurde von Edsger W. Dijkstra formuliert
- Die vorgestellte C++-Lösung illustriert:
  - Nebenläufigkeit durch Threads
  - Synchronisation über Mutexe
  - verklemmungsfreies Sperren
- Moderne C++-Sprachversion vereinfacht Programmierung gegenüber Low-Level-API auf Betriebssystemebene (z.B. pthreads)

Weitere Formen der Thread-Interaktion

- Bisher behandelt:
  - Mutexe und Locks
  - Atomare Variablen
- Typischer Anwendungsfall: Warten auf ein Ereignis / Setzen eines Flags

```
1  bool ready;
2  std::mutex mtx;
3
4  // Warten ...
5  std::unique_lock<std::mutex> l(mtx);
6  while (!ready) {
7      l.unlock();
8      std::this_thread::sleep_for(std::chrono::milliseconds(200));
9      l.lock();
10 }
```


Bedingungsvariablen

- Manchmal kann ein kritischer Abschnitt nur dann sinnvoll bearbeitet werden, wenn zusätzlich zu dem Erlangen eines Mutex auch noch eine weitere Bedingung erfüllt ist:
  - Erfordert synchronisierten Zugriff über `std::mutex` und `std::unique_lock`, die mit einer Bedingungsvariable vom Typ `std::condition_variable` „verknüpft“ werden
  - Thread wartet, bis die Bedingung erfüllt ist, und lässt während des Wartens den Mutex frei
  - Erfüllung der Bedingung wird durch anderen Thread mittels `notify_one` angezeigt ⇝ „Aufwecken“ eines wartenden Threads

```
1  std::mutex mtx;
2  std::unique_lock<std::mutex> l(mtx); // deferred lock!
3  std::condition_variable cond;
4  cond.wait(l, [&] {/* check condition */}); // Equiv. To:
5  while (![&] {/* check condition */}) {wait(l);}
```


Bedingungsvariablen: Beispiel (1)

```
1  std::list<std::string> shared_space;
2  std::mutex mtx;
3  std::condition_variable cond;
4
5  void consume() {
6      while (true) {
7          std::unique_lock<std::mutex> l(mtx); // deferred
8          cond.wait(l, [&] {
9              return !shared_space.empty();
10         }); // Nach wait() lock auf mtx für diesen Thread
11     std::string data = shared_space.front();
12     shared_space.pop_front();
13     l.unlock();
14     // data verarbeiten
15     }
16 }
```


Bedingungsvariablen: Beispiel (2)

```
1  void produce() {
2      // data erzeugen
3      std::lock_guard<std::mutex> lg(mtx);
4      shared_space.push_back(data);
5      cond.notify_one(); // OS weckt wartenden Thread
6  }
```


Thread-sichere Datenstrukturen

- Thread-Sicherheit := Eine Komponente kann gleichzeitig von verschiedenen Programmbereichen (Threads) mehrfach ausgeführt werden, ohne dass diese sich gegenseitig behindern
- Verschiedene Varianten:
  - Standard-Datenstruktur + über Mutexe/Sperren synchronisierte Zugriffe
  - Integration der Sperren in die Datenstruktur
  - Sperr-freie Datenstrukturen: nicht-blockierend, Vermeidung von Sperren, z.B. durch Compare/Exchange-Operationen

Thread-sichere Datenstrukturen: Anforderungen

- Mehrere Threads können gleichzeitig auf die Datenstruktur zugreifen
- Kein Thread sieht (Zwischen-)Zustand, bei dem Invarianten der Datenstruktur durch anderen Thread (kurzzeitig) verletzt sind
- Vermeidung von Race Conditions
- Vermeidung von Verklemmungen
- Korrekte Behandlung von Ausnahmen (Fehlern)

Beispiel: Thread-sichere Queue

```
1   template <typename T> class ts_queue {
2       private:
3       mutable std::mutex mtx;
4       std::condition_variable cond;
5       std::queue<T> the_queue;
6
7       public:
8       ts_queue() {}
9       ...
10  };
```

- Zeilen 1, 5: Kapselung der Klasse `std::queue`
- Zeile 3: Mutex für exklusiven Zugriff
- Zeile 4: Bedingungsvariable für Warten

Thread-sichere Queue: Methode push

```
1  void push(T val) {
2      std::lock_guard<std::mutex> l(mtx); // Immediate lock
3      the_queue.push(std::move(val));
4      cond.notify_one();
5  }
```

- Zeile 2: Lock Guard sichert exklusiven Zugriff
- Zeile 3: Element an die Queue anhängen
- Zeile 4: Aufwecken von eventuell wartenden Threads

Thread-sichere Queue: Methode waiting_pop

```
1  void waiting_pop(T& val) {
2    std::unique_lock<std::mutex> l(mtx); // Deferred lock
3    cond.wait(l, [this] { return !the_queue.empty(); });
4    val = std::move(the_queue.front());
5    the_queue.pop();
6  }
```

- Zeile 2: Unique Lock sichert exklusiven Zugriff
- Zeile 3: Warten bis Queue nicht mehr leer ist
- Zeilen 4, 5: Erstes Element aus der Queue entnehmen

Async, Future & Promise

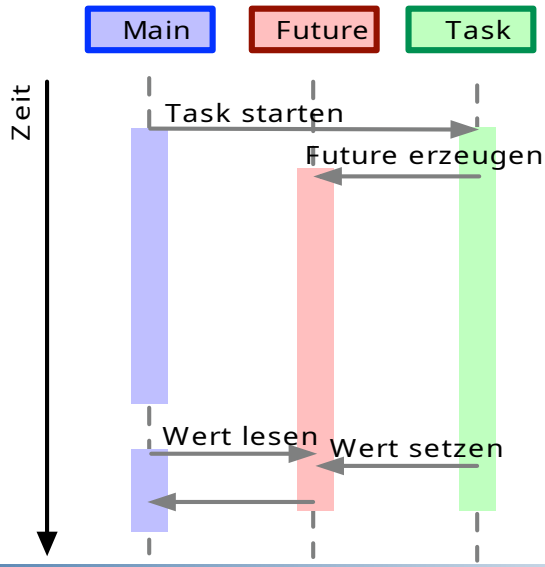
- `std::future` – Resultat einer asynchronen Berechnung, d.h. einer Berechnung, die erst noch stattfindet
- `std::async()` – asynchrones Starten eines Tasks

```
1  int long_calculation() {...}
2
3  std::future<int> result = std::async(long_calculation);
4  // Fortsetzung der Berechnung ...
5  result.wait();
6  std::cout << result.get() << std::endl;
```

- `std::promise` – erlaubt Wert zu setzen, wenn der aktuelle Thread beendet ist; oft in Kombination mit `std::future` eingesetzt
- future = Ergebnisobjekt, promise = Ergebnisproduzent

Future

- Methoden zum
  - Warten auf Ende des Tasks: `wait()`, `wait_for()`
  - Ergebnis lesen: `get()`



Future & Promise: Beispiel

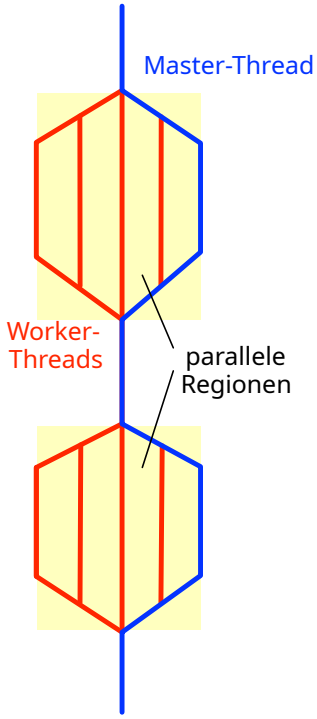
```
1  // Promise für einen String-Wert
2  std::promise<std::string> promise;
3  // zugehöriges Future-Objekt
4  auto res = promise.get_future();
5
6  // Produzenten-Thread
7  auto producer = std::thread([&] {
8  promise.set_value("Hello World!");
9  });
10 // Konsumenten-Thread
11 auto consumer = std::thread([&] {
12 std::cout << res.get() << "\n";
13 });
14
15 producer.join();
16 consumer.join();
```


Deklarative Parallelisierung mit OpenMP

- Programmierschnittstelle für Parallelisierung in C/C++/Fortran
- Programmiersprachenerweiterung durch Direktiven
- In C/C++: `#pragma omp ...`
- Zusätzliche Bibliotheksfunktionen: `#include <omp.h>`
- Aktuelle Version 5.0
- Untersützung in gcc und clang (nicht Apple!)
  - Vollständig für Version 4.5, partiell für Version 5.0
  - Nutzung über Compiler-Flag `-fopenmp`
- Beschränkt auf Architekturen mit gemeinsamen Speicher

OpenMP: Programmiermodell

- Master-Thread und mehrere Worker-Threads (Anzahl typischerweise durch OpenMP-Laufzeitsystem bestimmt)
- Über die `parallel`-Direktive kann Arbeit in einem Programmabschnitt auf Worker-Threads aufgeteilt werden
- Ende des parallelen Abschnitts ⇝ implizite Synchronisation
- Fortsetzung des Master-Threads



OpenMP: Hello World Programm

- Der dem `#pragma omp parallel` folgende Block wird parallel von allen Threads ausgeführt

```
1   #include <iostream>
2   #include <omp.h>
3
4   int main() {
5    #pragma omp parallel
6    { std::cout << "Hello World from thread #"
7                << omp_get_thread_num() << " of "
8                << omp_get_num_threads() << "\n";
9    }
10   std::cout << "Finished!\n";
11   return 0;
12  }
```


OpenMP: Schleifenparallelisierung

- Parallele Ausführung einer Schleife: jedem Thread wird ein Teil der Iterationen zugewiesen
- Für `for`-Schleifen mit:
  - eingeschränkter Syntax (ganzzahlige Schleifenvariablen, Operatoren auf Schleifenvariablen) und
  - für STL-Iteratoren (STL = Standard Template Library)

```
1  unsigned int results[20];
2
3  #pragma omp parallel for
4  for (int i = 0; i < 20; i++) {
5       int f = rand() % 30;
6       results[i] = fibonacci(f);
   }
```


OpenMP: Beeinflussung der Thread-Anzahl

- Maximale Anzahl:

```
1  unsigned int results[20];
2  #pragma omp parallel for num_threads(8)
3  for (int i = 0; i < 20; i++) {
4      results[i] = fibonacci(rand() % 30);
5  }
```

- Bedingte Parallelisierung:

```
1  std::vector<unsigned int> fibs(int max) {
2  std::vector<unsigned int> results(max);
3  #pragma omp parallel for if (max > 20)
4  for (int i = 0; i < max; i++)
5      results[i] = fibonacci(rand() % 30);
6  return results;
7  }
```


Aufteilung des Iterationsbereichs

- Der Iterationsbereich kann auf verschiedene Weisen auf Threads aufgeteilt werden
- Beeinflussung durch `schedule`-Direktive:
  - `schedule(auto)`: Default – implementierungsspezifisch
  - `schedule(static, n)`: statische Round-Robin-Verteilung – Bereiche der Größe n (Angabe von n ist optional)
  - `schedule(dynamic, n)`: dynamische Verteilung nach Bedarf
  - `schedule(guided, n)`: Verteilung nach Bedarf und proportional zur Restarbeit
  - ...

Geschachtelte Schleifen

- Parallelisierung mit `parallel for` beeinflusst nur die äußere Schleife
- `collapse(n)` gibt an, dass n Schleifen in einem gemeinsamen Iterationsbereich zusammengefasst und auf Threads verteilt werden sollen
- Beispiel Matrixmultiplikation:

```
1  #pragma omp parallel for collapse(3)
2  for (int row = 0; row < m; row++)
3      for (int col = 0; col < n; col++)
4          for (int inner = 0; inner < k; inner++)
5              prod[row][col] += A[row][inner] * B[inner][col];
```


Synchronisation

- Direktiven für parallele Ausführung:
  - `#pragma omp single/master` Abschnitt wird nur durch einen/den Master-Thread ausgeführt
  - `#pragma omp critical` kritischer Abschnitt
  - #pragma omp barrier Warten auf alle Worker-Threads
  - `#pragma omp atomic` kritischer Abschnitt - Zugriff auf gemeinsame Variable (z.B. Zähler)
- Speicherklauseln für Variablen:
  - `shared` für alle Threads sichtbar/änderbar
  - `private` jeder Thread hat eigene Kopie der Daten, wird nicht außerhalb initialisiert
  - `reduction` private Daten, die am Ende des Abschnitts zu globalem Wert zusammengefasst werden
  - `firstprivate/lastprivate` privat – initialisiert mit letztem Wert vor dem Abschnitt / Wert des letzten Threads der Iteration wird zurückgegeben

OpenMP: Parallele Abschnitte

- Zuweisung von Programmabschnitten zu Threads ⇝ statische Parallelität
- Geeignet z.B. für rekursive Abschnitte

```
1  void qsort(int data[], int left, int right) {
2      if (left < right) {
3          int p = partition(data, left, right);
4
5          #pragma omp parallel sections
6          {
7            #pragma omp section
8            qsort(data, left, p – 1);
9            #pragma omp section
10           qsort(data, p + 1, right);
11         }
12     }
13 }
```


OpenMP: Task-Programmierung

- Seit OpenMP 3.0 werden Tasks unterstützt, die:
  - automatisch Threads zugewiesen werden
  - an beliebiger Stelle definiert werden können
  - von beliebigen Threads definiert werden können

```
1  unsigned int fibonacci(unsigned int f) {
2      if (f < 2) return n;
3      unsigned int f1, f2;
4      #pragma omp task shared(f1)
5      f1 = fib(f - 1);
6      #pragma omp task shared(f2)
7      f2 = fib(f - 2);
8      #pragma omp taskwait
9      return f1 + f2;
10 }
```


Zusammenfassung

- C++ bietet weitreichende und mächtige Konzepte zur Parallelisierung:
  - Basiskonstrukte wie **Threads** und **Synchronisationsprimitive** (u.a. Mutexe)
  - Höherwertige Abstraktionen wie **async**, **Futures** und **Promises** ...
  - Deklarative Ansätze wie **OpenMP**
- Alle Formen von Parallelität (Instruktions-, Daten- und Taskparallelität) werden unterstützt
- Zusätzliche Bibliotheken und Frameworks wie Parallel STL, TBB, … bieten weitere Unterstützung
- Aber: Programmierung und Fehleranalyse sind erheblich anspruchsvoller als für herkömmliche (nicht-parallele) Software