philo.cpp

```
int main() {
   constexpr unsigned int PHILOSOPHERS = 5;
   std::thread philosopher[PHILOSOPHERS];
   std::mutex fork[PHILOSOPHERS];
   for (int i = 0; i < PHILOSOPHERS; ++i) {
      philosopher[i] = std::thread(Philosopher(i+1,
         fork[i], fork[(i + PHILOSOPHERS - 1) % PHILOSOPHERS]));
   }
   for (int i = 0; i < PHILOSOPHERS; ++i) {
      philosopher[i].join();
   }
}
```

- Die Philosophen können auf Basis von Threads leicht umgesetzt werden, wobei die Gabeln als Mutex-Variablen repräsentiert werden.

philo.cpp

```
class Philosopher {
   public:
      Philosopher(unsigned int id, std::mutex& left_fork,
            std::mutex& right_fork) :
         id(id), left_fork(left_fork), right_fork(right_fork)
      {
      }
      void operator()() {
         /* ... */
      }
   private:
      void print_status(const std::string& msg) const {
         std::lock_guard<std::mutex> lock(cout_mutex);
         std::cout << "philosopher [" << id << "]: " << msg << std::endl;
      }
      unsigned int id;
      std::mutex& left_fork;
      std::mutex& right_fork;
};
```

- Dabei erhält jeder Philosoph Referenzen auf die beiden Gabeln.

philo.cpp

```
    void operator()() {
       for (int i = 0; i < 5; ++i) {
          print_status("sits down at the table");
          {
             std::lock_guard<std::mutex> lock1(left_fork);
             print_status("picks up the left fork");
             {
                std::lock_guard<std::mutex> lock2(right_fork);
                print_status("picks up the right fork");
                {
                   print_status("is dining");
                }
             }
             print_status("returns the right fork");
          }
          print_status("returns the left fork");
          print_status("leaves the table");
       }
    }
```

- Alle Philosophen nehmen hier zunächst die linke und dann die rechte Gabel. Bei dieser Vorgehensweise sind bekanntlich Deadlocks nicht ausgeschlossen.

$$P = (P_1 \,|||\, P_2) \,||\, MX_1 \,||\, MX_2$$
$$MX_1 = (lock_1 \rightarrow unlock_1 \rightarrow MX_1)$$
$$MX_2 = (lock_2 \rightarrow unlock_2 \rightarrow MX_2)$$
$$P_1 = (lock_1 \rightarrow lock_2 \rightarrow unlock_2 \rightarrow unlock_1 \rightarrow P_1)$$
$$P_2 = (lock_2 \rightarrow lock_1 \rightarrow unlock_1 \rightarrow unlock_2 \rightarrow P_2)$$

- Wenn P_1 und P_2 beide sofort jeweils ihren ersten Lock erhalten, gibt es einen Deadlock.

- Eine der von Dijkstra vorgeschlagenen Deadlock-Vermeidungsstrategien (siehe EWD625) sieht die Definition einer totalen Ordnung aller MX_i vor, die z.B. durch die Indizes zum Ausdruck kommen kann.
- D.h. wenn MX_i und MX_j gehalten werden sollen, dann ist zuerst MX_i zu belegen, falls $i < j$, ansonsten MX_j.
- Dijkstra betrachtet den Ansatz selbst als hässlich, weil damit eine willkürliche Anordnung erzwungen wird. Die Vorgehensweise ist nicht immer praktikabel, aber nicht selten eine sehr einfach umzusetzende Lösung.

philo2.cpp

```
struct Fork {
   unsigned id; std::mutex mutex;
};
/* ... */
int main() {
   constexpr unsigned int PHILOSOPHERS = 5;
   std::thread philosopher[PHILOSOPHERS];
   Fork fork[PHILOSOPHERS];
   for (int i = 0; i < PHILOSOPHERS; ++i) {
      fork[i].id = i;
   }
   for (int i = 0; i < PHILOSOPHERS; ++i) {
      philosopher[i] = std::thread(Philosopher(i+1,
         fork[i], fork[(i + PHILOSOPHERS - 1) % PHILOSOPHERS]));
   }
   for (int i = 0; i < PHILOSOPHERS; ++i) {
      philosopher[i].join();
   }
}
```

- Um das umzusetzen, werden allen Ressourcen (hier den Gabeln) eindeutige Nummern vergeben.

philo2.cpp

```
std::mutex* first; std::mutex* second;
std::string first_text; std::string second_text;
if (left_fork.id < right_fork.id) {
   first = &left_fork.mutex; second = &right_fork.mutex;
   first_text = "left"; second_text = "right";
} else {
   first = &right_fork.mutex; second = &left_fork.mutex;
   first_text = "right"; second_text = "left";
}
{
   std::lock_guard<std::mutex> lock1(*first);
   print_status("picks up the " + first_text + " fork");
   {
      std::lock_guard<std::mutex> lock2(*second);
      print_status("picks up the " + second_text + " fork");
      {
         print_status("is dining");
      }
   }
   print_status("returns the " + second_text + " fork");
}
print_status("returns the " + first_text + " fork");
print_status("leaves the table");
```

- C++ bietet zwei Klassen an, um Locks zu halten: *std*::*lock_guard* und *std*::*unique_lock*.
- Beide Varianten unterstützen die automatische Freigabe durch den jeweiligen Dekonstruktor.
- In einfachen Fällen, bei denen es nur um den gegenseitigen Ausschluss geht, wird *std*::*lock_guard* verwendet.
- *std*::*unique_lock* bietet mehr Möglichkeiten. Dazu gehören insbesondere folgende Optionen:

std::defer_lock	der Mutex wird noch nicht belegt
std::try_to_lock	es wird nicht-blockierend versucht, den Mutex zu belegen
std::adopt_lock	ein bereits belegter Mutex wird übernommen

- Einige weitere Klassen wie u.a. *std*::*shared_mutex* und *std*::*shared_lock* sind erst in C++14 aufgenommen worden.

philo3.cpp

```
{
   std::unique_lock<std::mutex> lock1(left_fork, std::defer_lock);
   std::unique_lock<std::mutex> lock2(right_fork, std::defer_lock);
   std::lock(lock1, lock2);
   print_status("picks up both forks and is dining");
}
```

- C++ bietet mit *std::lock* eine Operation an, die beliebig viele Mutex-Variablen beliebigen Typs akzeptiert, und diese in einer vom System gewählten Reihenfolge belegt, die einen Deadlock vermeidet.
- Normalerweise erwartet *std::lock* Mutex-Variablen. *std::unique_lock* ist eine Verpackung, die wie eine Mutex-Variable verwendet werden kann.
- Zunächst nehmen die beiden *std::unique_lock* die Mutex-Variablen jeweils in Beschlag, ohne eine *lock*-Operation auszuführen (*std::defer_lock*). Danach werden nicht die originalen Mutex-Variablen, sondern die *std::unique_lock*-Objekte an *std::lock* übergeben.
- Diese Variante ist umfassend auch gegen Ausnahmenbehandlungen abgesichert.

- Ein Monitor ist eine Klasse, bei der maximal ein Thread eine Methode aufrufen kann.
- Wenn weitere Threads konkurrierend versuchen, eine Methode aufzurufen, werden sie solange blockiert, bis sie alleinigen Zugriff haben (gegenseitiger Ausschluss).
- Der Begriff und die zugehörige Idee gehen auf einen Artikel von 1974 von C. A. R. Hoare zurück.
- Aber manchmal ist es sinnvoll, den Aufruf einer Methode von einer weiteren Bedingung abhängig zu machen,

- Bei Monitoren können Methoden auch mit Bedingungen versehen werden, d.h. eine Methode kommt nur dann zur Ausführung, wenn die Bedingung erfüllt ist.
- Wenn die Bedingung nicht gegeben ist, wird die Ausführung der Methode solange blockiert, bis sie erfüllt ist.
- Eine Bedingung sollte nur von dem internen Zustand eines Objekts abhängen.
- Bedingungsvariablen sind daher private Objekte eines Monitors mit den Methoden *wait*, *notify_one* und *notify_all*.
- Bei *wait* wird der aufrufende Thread solange blockiert, bis ein anderer Thread bei einer Methode des Monitors *notify_one* oder *notify_all* aufruft. (Bei *notify_all* können alle, die darauf gewartet haben, weitermachen, bei *notify_one* nur ein Thread.)
- Eine Notifizierung ohne darauf wartende Threads ist wirkungslos.

```
class Monitor {
   public:
      void some_method() {
         std::unique_lock<std::mutex> lock(mutex);
         while (! /* some condition */) {
            condition.wait(lock);
         }
         // ...
      }
      void other_method() {
         std::unique_lock<std::mutex> lock(mutex);
         // ...
         condition.notify_one();
      }
   private:
      std::mutex mutex;
      std::condition_variable condition;
};
```

- Bei der C++11-Standardbibliothek ist eine Bedingungsvariable immer mit einer Mutex-Variablen verbunden.
- *wait* gibt den Lock frei, wartet auf die Notifizierung, wartet dann erneut auf einen exklusiven Zugang und kehrt dann zurück.

Verknüpfung von Bedingungs- und Mutex-Variablen 111

- Die Methoden *notify_one* oder *notify_all* sind wirkunslos, wenn kein Thread auf die entsprechende Bedingung wartet.
- Wenn ein Thread feststellt, dass gewartet werden muss und danach wartet, dann gibt es ein Fenster zwischen der Feststellung und dem Aufruf von *wait*.
- Wenn innerhalb des Fensters *notify_one* oder *notify_all* aufgerufen wird, bleiben diese Aufrufe wirkungslos und beim anschließenden *wait* kann es zu einem Deadlock kommen, da dies auf eine Notifizierung wartet, die nun nicht mehr kommt.
- Damit das Fenster völlig geschlossen wird, muss *wait* als atomare Operation zuerst den Thread in die Warteschlange einreihen und erst dann den Lock freigeben.
- Bei *std::condition_variable* muss der Lock des Typs *std*::*unique_lock*<*std*::*mutex*> sein. Für andere Locks gibt es die u.U. weniger effiziente Alternative *std::condition_variable_any*.

$$
\begin{aligned}
& M \,||\, (P_1 \,|||\, P_2 \,|||\, CL) \,||\, C \\
M &= (lock \rightarrow unlock \rightarrow M) \\
P_1 &= (lock \rightarrow wait \rightarrow resume \rightarrow \\
& \quad critical\_region_1 \rightarrow unlock \rightarrow P_1) \\
P_2 &= (lock \rightarrow critical\_region_2 \rightarrow \\
& \quad (notify \rightarrow unlock \rightarrow P_2 \mid unlock \rightarrow P_2)) \\
CL &= (unlock_C \rightarrow unlock \rightarrow unlocked_C \rightarrow \\
& \quad lock_C \rightarrow lock \rightarrow locked_C \rightarrow CL) \\
C &= (wait \rightarrow unlock_C \rightarrow unlocked_C \rightarrow notify \rightarrow \\
& \quad lock_C \rightarrow locked_C \rightarrow resume \rightarrow C)
\end{aligned}
$$

- Einfacher Fall mit M für die Mutex-Variable, einem Prozess P_1, der auf eine Bedingungsvariable wartet, einem Prozess P_2, der notifiziert oder es auch sein lässt, und der Bedingungsvariablen C, die hilfsweise CL benötigt, um gemeinsam mit P_1 und P_2 um die Mutexvariable konkurrieren zu können.

- Wenn notwendig, können auch eigene Klassen für Locks definiert werden.
- Die Template-Klasse *std::lock_guard* akzeptiert eine beliebige Lock-Klasse, die mindestens folgende Methoden unterstützt:

void *lock*()	blockiere, bis der Lock reserviert ist
void *unlock*()	gib den Lock wieder frei

- Typhierarchien und virtuelle Methoden werden hierfür nicht benötigt, da hier statischer Polymorphismus vorliegt, bei dem mit Hilfe von Templates alles zur Übersetzzeit erzeugt und festgelegt wird.
- In einigen Fällen (wie etwa die Übergabe an *std::lock*) wird auch noch folgende Methode benötigt:

bool *try_lock*()	versuche, nicht-blockierend den Lock zu reservieren

Locks für Resourcen mit einer Kapazität

resource-lock.hpp

```
class ResourceLock {
   public:
      ResourceLock(unsigned int capacity) : capacity(capacity), used(0) {
      }
      void lock() {
         std::unique_lock<std::mutex> lock(mutex);
         if (used == capacity) {
            released.wait(lock);
         }
         ++used;
      }
      void unlock() {
         std::unique_lock<std::mutex> lock(mutex);
         assert(used > 0);
         --used;
         released.notify_one();
      }
   private:
      const unsigned int capacity;
      unsigned int used;
      std::mutex mutex;
      std::condition_variable released;
};
```

philo4.cpp

```
constexpr unsigned int PHILOSOPHERS = 5;
ResourceLock rlock(PHILOSOPHERS-1);
std::thread philosopher[PHILOSOPHERS];
std::mutex fork[PHILOSOPHERS];
for (int i = 0; i < PHILOSOPHERS; ++i) {
   philosopher[i] = std::thread(Philosopher(i+1,
      fork[i], fork[(i + PHILOSOPHERS - 1) % PHILOSOPHERS], rlock));
}
```

- Mit Hilfe eines *ResourceLock* lässt sich das Philosophenproblem mit Hilfe eines Dieners lösen.
- Bei n Philosophen lassen die Diener zu, dass sich $n-1$ Philosophen hinsetzen.

Philosophen mit Diener

philo4.cpp

```
void operator()() {
   for (int i = 0; i < 5; ++i) {
      print_status("comes to the table");
      {
         std::lock_guard<ResourceLock> lock(rlock);
         print_status("got permission to sit down at the table");
         {
            std::lock_guard<std::mutex> lock1(left_fork);
            print_status("picks up the left fork");
            {
               std::lock_guard<std::mutex> lock2(right_fork);
               print_status("picks up the right fork");
               {
                  print_status("is dining");
               }
            }
            print_status("returns the right fork");
         }
         print_status("returns the left fork");
      }
      print_status("leaves the table");
   }
}
```

- C++ bietet mit *std::promise* und *std::future* eine einfache Kommunikations- und Synchronisierungsmöglichkeit an.
- Mit *std*::*promise*<*T*> *p*; wird ein „Versprechen“ gegeben, irgendwann einmal mit *p.set_value*(*value*) einen Wert zu setzen.
- Mit *std*::*future*<*T*> $f = p.get\_future()$ darf daraus genau einmal(!) ein zugehöriges *std::future*-Objekt abgeleitet werden.
- Typischerweise „gehören“ dann die beiden Objekte unterschiedlichen Threads.
- Mit *f.get*() wartet der Aufrufer darauf, dass der Wert mit *p.set_value*(*value*) gesetzt wird und liefert dann diesen zurück.
- Normale Zuweisungen sind weder für *std::promise*- noch *std::future*-Objekte zulässig. Ein Paar dient dazu, einen Wert genau einmal synchronisert weiterzugeben.

simpson-fp.cpp

```
double mt_simpson(double (*f)(double), double a, double b, int n,
      int nofthreads) {
   // divide the given interval into nofthreads partitions
   assert(n > 0 && a <= b && nofthreads > 0);
   int nofintervals = n / nofthreads; int remainder = n % nofthreads;
   int interval = 0;

   std::future<double> results[nofthreads];
   // ... fork ...
   // join threads and sum up their results
   double sum = 0;
   for (int i = 0; i < nofthreads; ++i) {
      sum += results[i].get();
   }
   return sum;
}
```

- Das Array der hier deklarierten *std::future*-Objekte ist zu Beginn noch „leer“, d.h. die Objekte sind noch nicht mit *std::promise*-Objekten verbunden.
- Am Ende werden die Ergebnisse eingesammelt. Die Synchronisierung erfolgt hier implizit bei der *get*-Methode.

simpson-fp.cpp

```
// fork off the individual threads
double x = a;
for (int i = 0; i < nofthreads; ++i) {
   int intervals = nofintervals;
   if (i < remainder) ++intervals;
   interval += intervals;
   double xleft = x; x = a + interval * (b - a) / n;
   std::promise<double> promise;
   results[i] = promise.get_future();
   auto t = std::thread([=, promise=std::move(promise)]() mutable {
      promise.set_value(simpson(f, xleft, x, intervals));
   });
   t.detach(); // no longer joinable, t object can be destructed
}
```

- In der Schleife werden lokal temporäre *std::promise*-Objekte erzeugt und daraus jeweils das zugehörige *std::future*-Objekt abgeholt.
- Das *std::future*-Objekt wird in *results* abgespeichert, das *std::promise*-Objekt geht an den zu erzeugenden Thread.

simpson-fp.cpp

```
std::promise<double> promise;
results[i] = promise.get_future();
auto t = std::thread([=, promise = std::move(promise)]() mutable {
   promise.set_value(simpson(f, xleft, x, intervals));
});
t.detach(); // no longer joinable, t object can be destructed
```

- Hier wird zunächst ein *std::promise*-Objekt lokal erzeugt und von diesem ein *std::future*-Objekt abgeleitet, das in dem *results*-Array abgespeichert wird.
- Dann wird ein Thread gestartet, bei dem als Funktions-Objekt ein Lambda-Ausdruck übergeben wird.
- Der Lambda-Ausdruck übernimmt diverse Parameter implizit durch Zuweisung und im Falle von *promise* durch *std::move*, da eine normale Zuweisung nicht zulässig ist. (Dies geht so erst ab C++14, bei C++11 wird noch *std::bind* benötigt.)
- Ohne **mutable** kann der Lambda-Ausdruck *promise* nicht verändern.

simpson-fp.cpp

```
t.detach(); // no longer joinable, t object can be destructed
```

- Auf einen Thread muss entweder mit *join* irgendwann gewartet werden oder dieser muss explizit mit *detach* „vergessen“ werden. Nach *detach* ist eine Synchronisierung mit *join* nicht mehr möglich. Das ist hier auch nicht notwendig, da die Synchronisierung über die *std::future*-Objekte erfolgt.
- Wenn ein Thread-Objekt dekonstruiert wird, für den weder *join* noch *detach* aufgerufen wurde, dann wird die gesamte Programmausführung gewaltsam mit *std::terminate* beendet.

simpson-async.cpp

```
results[i] = std::async([=]() -> double {
   return simpson(f, xleft, x, intervals);
});
```

- Den Standard-Fall, dass ein Thread genau einen Wert berechnet, der über ein *std::future*-Objekt synchronisiert abgeholt werden kann, wird mit *std::async* vereinfacht.
- *std::async* benötigt ein Funktionsobjekt (hier ein Lambda-Ausdruck), das den gewünschten Wert mit **return** zurückliefert.
- *std::async* erzeugt ein *std::promise*-Objekt, leitet davon das *std::future*-Objekt ab, das zurückgegeben wird, ruft dann in einem neu erzeugten Thread das Funktionsobjekt auf und weist den zurückgegebenen Wert dem *std::promise*-Objekt zu.