$$
\begin{aligned}
MX &= M \parallel (P_1 \,|||\, \ldots \,|||\, P_n) \\
M &= (lock \rightarrow unlock \rightarrow M) \\
P_i &= (lock \rightarrow begin\_critical\_region_i \rightarrow \\
&\quad end\_critical\_region_i \rightarrow unlock \rightarrow \\
&\quad private\_work_i \rightarrow P_i)
\end{aligned}
$$

- n Prozesse $P_1 \ldots P_n$ wollen konkurrierend auf eine Ressource zugreifen, aber zu einem Zeitpunkt soll diese nur einem Prozess zur Verfügung stehen.
- Die Ressource wird von M verwaltet.
- Wegen dem $|||$-Operator wird unter den P_i jeweils ein Prozess nicht-deterministisch ausgewählt, der für *lock* bereit ist.

queens.cpp

```
#include <mutex>

std::mutex cout_mutex;

// ...

void print_board() {
   std::lock_guard<std::mutex> lock(cout_mutex);
   // cout << ...
}
```

- Bei Threads wird ein gegenseitiger Ausschluss über sogenannte Mutex-Variablen (*mutual exclusion*) organisiert.
- Sinnvollerweise sollten Ausgaben auf den gleichen Stream (hier *std::cout*) synchronisiert erfolgen.
- Deswegen darf nur der Thread schreiben, der exklusiven Zugang zu der Mutex-Variable *cout_mutex* hat.
- Mit *std::lock_guard* gibt es die Möglichkeit, ein Objekt zu erzeugen, das den Mutex bei der Konstruktion reserviert und bei der Dekonstruktion automatisiert freigibt.

queens.cpp

```
#include <mutex>

std::mutex cout_mutex;

// ...

void print_board() {
   std::lock_guard<std::mutex> lock(cout_mutex);
   // cout << ...
}
```

- Code-Bereiche, die nur im gegenseitigen Ausschluss ausgeführt werden können, werden kritische Regionen genannt (*critical regions*).
- Am einfachsten ist es, beim Zugriff auf eine Datenstruktur alle anderen auszuschließen.
- Gelegentlich ist der gegenseitige Ausschluss auch feiner granuliert, wenn sich Teile der gemeinsamen Datenstruktur unabhängig voneinander ändern lassen. Dieser Ansatz führt zu weniger Behinderungen, ist aber auch fehleranfälliger.

queens2.cpp

```
class Results {
   public:
      void add(const PositionList& result) {
         std::lock_guard<std::mutex> lock(mutex);
         results.insert(result);
      }
      void print() const {
         std::lock_guard<std::mutex> lock(mutex);
         for (const PositionList& result: results) {
            print_board(result);
         }
      }
   private:
      mutable std::mutex mutex;
      std::set<PositionList> results;
};
```

- Bei der Klasse *Results* sind alle Methoden in kritischen Regionen.
- Das ermöglicht die statische Überprüfung, dass keine konkurrierenden Zugriffe auf die Datenstruktur stattfinden und die Freigabe des Locks nirgends vergessen wird.

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 ihm zugeordneten 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 ist ein Deadlock möglich, wenn beispielsweise alle gleichzeitig die linke Gabel nehmen.

$$P = (P_1 \,|||\, P_2) \parallel MX_1 \parallel MX_2$$
$$MX_1 = (lock_1 \rightarrow unlock_1 \rightarrow MX_1)$$
$$MX_2 = (lock_2 \rightarrow unlock_2 \rightarrow MX_2)$$
$$P_1 = (lock_1 \rightarrow think_1 \rightarrow lock_2 \rightarrow unlock_2 \rightarrow unlock_1 \rightarrow P_1)$$
$$P_2 = (lock_2 \rightarrow think_2 \rightarrow lock_1 \rightarrow unlock_1 \rightarrow unlock_2 \rightarrow P_2)$$

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

$$\nexists\, t : t \in traces(P) \wedge t > \langle lock_1, think_1, lock_2, think_2 \rangle$$

- 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 unschön, 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 (das sind hier die 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.
- *std*::*lock_guard* ist eine reine RAII-Klasse: Die Ressource wird nur bei der Konstruktion belegt und die Freigabe erfolgt ausschließlich durch den Abbau.
- *std*::*unique_lock* bietet u.a. die Methoden *lock* und *unlock* an und einige spezielle Optionen bei der Konstruktion:

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* werden wohl ab C++17 unterstützt werden.

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 118

- 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*.

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

- 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

resource-lock.hpp

```
class ResourceLock {
   public:
      ResourceLock(unsigned int capacity) : capacity(capacity), used(0) {
      }
      void lock() {
         std::unique_lock<std::mutex> lock(mutex);
         while (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;
};
```

Philosophen mit Diener

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");
   }
}
```