- 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

- Bei der Boost-Bibliothek gab es noch mehr solcher Klassen, u.a. *boost*::*shared_lock*, die jedoch nicht mehr in C++11 aufgenommen worden sind.

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, bleibt dieses 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");
   }
}
```

- Ringpuffer sind FIFO-Queues auf Basis eines festdimensionierten Arrays.
- Die Pipelines unter UNIX werden intern auf Basis von Ringpuffern realisiert.
- Typischerweise hat ein Ringpuffer die Methoden *read* und *write*.
- Die Methode *read* hat als Vorbedingung, dass es etwas zu lesen gibt, d.h. der Puffer darf nicht leer sein.
- Die Methode *write* hat als Vorbedingung, dass es noch Platz zum Schreiben gibt, d.h. der Puffer darf nicht vollständig gefüllt sein.

ringbuffer.hpp

```
#include <vector>
#include <mutex>
#include <condition_variable>

template<typename T>
class RingBuffer {
   public:
      RingBuffer(unsigned int size) : /* ... */ {}
      void write(T item) { /* ... */ }
      void read(T& item) { /* ... */ }
      }
   private:
      unsigned int read_index; unsigned int write_index;
      unsigned int filled;
      std::vector<T> buf;
      std::mutex mutex;
      std::condition_variable ready_for_reading;
      std::condition_variable ready_for_writing;
};
```

- Die beiden Bedingungen werden durch *ready_for_reading* und *ready_for_writing* repräsentiert.

ringbuffer.hpp

```
RingBuffer(unsigned int size) :
   read_index(0), write_index(0), filled(0), buf(size) {
}
```

- Zu Beginn sind beide Indizes und der Füllgrad bei 0.
- Der Puffer wird in Abhängigkeit der gegebenen Größe dimensioniert.

ringbuffer.hpp

```
void write(T item) {
   std::unique_lock<std::mutex> lock(mutex);
   while (filled == buf.capacity()) {
      ready_for_writing.wait(lock);
   }
   buf[write_index] = item;
   write_index = (write_index + 1) % buf.capacity();
   ++filled;
   ready_for_reading.notify_one();
}
```

- Wenn der Puffer bereits voll ist, wird mit Hilfe der Bedingungsvariablen *ready_for_writing* darauf gewartet, dass wieder Kapazität frei wird.
- Prinzipiell könnte die **while**-Schleife durch eine **if**-Anweisung ersetzt werden. So ist der Programmtext jedoch etwas robuster.
- Wenn die Pufferkapazität erhöht wird, ist zu bedenken, dass möglicherweise ein anderer Thread darauf wartet.

Lesen von einem Ringbuffer

ringbuffer.hpp

```
void read(T& item) {
   std::unique_lock<std::mutex> lock(mutex);
   while (filled == 0) {
      ready_for_reading.wait(lock);
   }
   item = buf[read_index];
   read_index = (read_index + 1) % buf.capacity();
   --filled;
   ready_for_writing.notify_one();
}
```

- Dazu symmetrisch ist die Lese-Methode, die zu Beginn sicherstellt, dass es etwas zu Lesen gibt.

- Idee: Zu einer konkurierrend benutzten Datenstruktur gehört ein eigener Thread.
- Dieser greift alleine auf die Datenstruktur zu und kann somit auch nicht in Konflikt geraten.
- Andere Threads, die auf die Datenstruktur zugreifen wollen, müssen mit dem die Datenstruktur kontrollierenden Thread kommunizieren.
- Diese synchronen Begegnungen zwischen Threads nennen sich Rendezvous.
- Geprägt wurde der Begriff durch Ada, das sich in dieser Beziehung sehr an CSP anlehnte.

Ringpuffer in Ada nach Ichbiah


```
task BUFFERING is
   entry READ (V : out ITEM);
   entry WRITE(E : in  ITEM);
end;

task body BUFFERING is
   SIZE      : constant INTEGER := 10;
   BUFFER    : array (1 .. SIZE) of ITEM;
   INX, OUTX : INTEGER range 1 .. SIZE := 1;
   COUNT     : INTEGER range 0 .. SIZE := 0;
begin
   loop
      select
         when COUNT < SIZE =>
            accept WRITE(E : in  ITEM) do
               BUFFER(INX) := E;
            end;
            INX := INX mod SIZE + 1;
            COUNT := COUNT + 1;
      or
         when COUNT > 0 =>
            accept READ (V : out ITEM) do
               V := BUFFER(OUTX);
            end;
            OUTX := OUTX mod SIZE + 1;
            COUNT := COUNT - 1;
      end select;
   end loop;
end BUFFERING;
```

- Zwischen den beiden Threads existiert eine Kommunikations-Datenstruktur, auf die konkurrierend zugegriffen wird.
- (Ja, das wollen wir generell vermeiden, aber zur Umsetzung von Rendezvous wird dies genau einmal benötigt.)
- Diese Kommunikations-Datenstruktur nimmt Anfragen auf, die Methodenaufrufen entsprechen. Jedes dieser Objekte besteht aus einer Bezeichnung der Anfrage (*Entry*) und einem Objekt mit dem Parametern (*Request*).
- Bei einem Methodenaufruf wird dann so eine Anfrage erzeugt, in die Datenstruktur abgelegt und auf die Bearbeitung gewartet.
- Der andere Thread sucht sich dann gelegentlich eine der Anfragen aus, die bearbeitbar sind, bearbeitet sie und markiert sie dann als erledigt. Danach kann der anfragende Thread wieder aufgeweckt werden.

rv-ringbuffer.hpp

```
enum Entry {RingBufferRead, RingBufferWrite};
template<typename T>
struct Request {
   Request() : itemptr(0) {};
   Request(T* ip) : itemptr(ip) {};
   T* itemptr;
};
```

- Jede Methode entspricht ein Wert beim Aufzählungsdatentyp *Entry*. Bei einem Ringpuffer haben wir nur die Methoden *read* und *write*.
- Alle Parameter aller Methodenaufrufe werden in der Datenstruktur *Request* zusammengefasst. Das ist hier nur ein Zeiger auf ein Objekt, das in den Ringpuffer hineinzulegen oder aus diesem herauszunehmen ist.

rendezvous.hpp

```
struct Member {
   Member() {};
   Member(const Entry& e, const Request& r) :
      entry(e), req(r), done(new std::condition_variable()) {
   };
   Member(const Member& other) :
      entry(other.entry), req(other.req), done(other.done) {};
   Entry entry;
   Request req;
   std::shared_ptr<std::condition_variable> done;
};
```

- Eine Struktur des Datentyp *Member* fasst *Entry*, *Request* und eine Bedingungsvariable *done* zusammen.
- Die Bedingungsvariable signalisiert, wann die Anfrage erledigt ist.
- Da diese Datenstruktur in Container hinein- und herauskopiert wird, ist es notwendig, die Bedingungsvariable hinter einem Zeiger zu verstecken. Das Aufräumen übernimmt hier *std*::*shared_ptr*.

rendezvous.hpp

```
typedef std::list<Member> Queue;
typedef std::map<Entry, Queue> Requests;
Requests requests;
```

- Alle Anfragen, die die gleiche Methode (*Entry*) betreffen, werden in eine Warteschlange (*Queue*) zusammengefasst.
- Alle Warteschlangen sind über ein assoziatives Array zugänglich (*Requests*).
- Der konkurrierende Zugriff wird über *mutex* und *submitted* geregelt. Letzteres signalisiert das Hinzufügen einer Anfrage.

rendezvous.hpp

```
std::mutex mutex;
std::condition_variable submitted;
```

Einreichen einer Anfrage

rendezvous.hpp

```
template<typename Request, typename Entry>
class Rendezvous {
   public:
      // ...

      void connect(Entry entry, Request request) {
         Member member(entry, request);
         // submit request
         std::unique_lock<std::mutex> lock(mutex);
         requests[entry].push_back(member);
         submitted.notify_all();
         // wait for its completion
         member.done->wait(lock);
      }

   private:
      // ...
};
```

- *connect* wird zum Einreichen einer Anfrage verwendet: Nach dem Eintragen in die passende Warteschlange wird auf die Erledigung gewartet.

rv-ringbuffer.hpp

```
template< typename T, typename RT = Request<T> >
class RingBuffer: RendezvousTask<RT, Entry> {
   public:
      RingBuffer(unsigned int size) :
            read_index(0), write_index(0), filled(0), buf(size) {
         assert(size > 0);
         this->start(); /* start the associated thread */
      }
      void write(T item) {
         this->rv.connect(RingBufferWrite, RT(&item));
      }
      void read(T& item) {
         this->rv.connect(RingBufferRead, RT(&item));
      }

      // ...
   private:
      unsigned int read_index;
      unsigned int write_index;
      unsigned int filled;
      std::vector<T> buf;
};
```

rendezvous.hpp

```
Entry select(const EntrySet& entries) throw(TerminationException) {
   assert(entries.size() > 0); /* deadlock otherwise */
   std::unique_lock<std::mutex> lock(mutex);
   for(;;) {
      for (auto& entry: entries) {
         if (requests.find(entry) != requests.end()) {
            return entry;
         }
      }
      submitted.wait(lock);
      if (terminating) throw TerminationException();
   }
}
```

- *select* gehört zur Template-Klasse *Rendezvous* und erhält eine Menge akzeptabler Anfragen (*entries*).
- Es wird dann überprüft, ob so eine Anfrage vorliegt. Falls nicht, wird darauf gewartet.
- Die Variable *terminating* dient später dazu, den Thread kontrolliert wieder abzubauen.

Ablauf des Ringpuffer-Threads

rv-ringbuffer.hpp

```
template< typename T, typename RT = Request<T> >
class RingBuffer: RendezvousTask<RT, Entry> {
   public:
      // ...

      virtual void operator()() {
         for(;;) {
            // task body ...
         }
      }

   private:
      // ...
};
```

- Der für den Ringpuffer zuständige Thread wird von dem Funktionsoperator repräsentiert.
- Diese wird indirekt von der Template-Klasse *RendezvousTask* aufgerufen, die in der Methode *start* den Thread startet.

rv-ringbuffer.hpp

```
std::set<Entry> entries;
if (filled > 0) {
   entries.insert(RingBufferRead);
}
if (filled < buf.capacity()) {
   entries.insert(RingBufferWrite);
}
Entry entry = this->rv.select(entries);
switch (entry) {
   case RingBufferRead:
      // ...
      break;
   case RingBufferWrite:
      // ...
      break;
}
```

- Zuerst wird festgestellt, welche Anfragen zulässig sind und eine entsprechende Menge erstellt.
- Dann wird mit *select* auf das Eintreffen einer entsprechenden Anfrage gewartet bzw. sie ausgewählt.

rv-ringbuffer.hpp

```
case RingBufferRead:
   {
      RT request;
      typename Rendezvous<RT, Entry>::Accept(this->rv,
         RingBufferRead, request);
      *(request.itemptr) = buf[read_index];
      read_index = (read_index + 1) % buf.capacity();
      --filled;
   }
   break;
```

- Entscheidend ist die Frage, wann eine Anfrage erledigt ist.
- Dies ist hier durch die Lebensdauer des *Accept*-Objekts geregelt. Sobald dieses dekonstruiert wird, ist die Frage erledigt und der anfragende Thread kann weiterarbeiten.

rv-ringbuffer.hpp

```
case RingBufferWrite:
   {
      RT request;
      typename Rendezvous<RT, Entry>::Accept(this->rv,
         RingBufferWrite, request);
      buf[write_index] = *(request.itemptr);
      write_index = (write_index + 1) % buf.capacity();
      ++filled;
   }
   break;
```

- Das seltsame Konstrukt *this−>rv* an Stelle von *rv* (als Verweis auf das zugehörige Rendezvous-Objekt) ist notwendig, weil die Basisklasse von Template-Parametern abhängt und daher nur eingeschränkt sichtbar ist.

Herausnehmen einer Anfrage

rendezvous.hpp

```
class Accept {
   public:
      Accept(Rendezvous& rv, Entry entry, Request& request)
            throw(TerminationException) :
               lock(rv.mutex) {
         typename Requests::iterator it;
         for(;;) {
            it = rv.requests.find(entry);
            if (it != rv.requests.end()) break;
            rv.submitted.wait(lock);
            if (rv.terminating) throw TerminationException();
         }
         member = it->second.front();
         request = member.req;
         it->second.pop_front();
         if (it->second.empty()) {
            rv.requests.erase(it);
         }
      }
      ~Accept() {
         member.done->notify_all();
      }
   private:
      Member member;
      std::unique_lock<std::mutex> lock;
};
```

- Grundsätzlich ist der Abbau von Objekten mit zugehörigen Threads nicht-trivial.
- Die Threads-Bibliothek besteht darauf, dass
  - jedes *std*::*mutex*-Objekt beim Abbau ungelockt sein muss,
  - niemand auf eine *std*::*condition_variable* wartet und
  - *join* aufgerufen sein muss,
- Entsprechend muss allen beteiligten Parteien der Abbau signalisiert werden; diese müssen darauf reagieren (z.B. durch Ausnahmenbehandlungen) und es muss darauf gewartet werden, dass dies alles abgeschlossen ist.

rendezvous.hpp

```
struct TerminationException: public std::exception {
   public:
      virtual const char* what() const throw() {
         return "thread has been terminated";
      }
};
```

- Für die Ausnahmenbehandlung zur Terminierung wird sinnvollerweise eine eigene Klasse definiert.
- Hier erfolgt dies als Erweiterung von *std*::*exception*, wobei die Methode *what* überdefiniert wird, die einen lesbaren Text zurückliefert.

Basisklasse für Rendezvous-Threads

rendezvous.hpp

```
template<typename Request, typename Entry>
class RendezvousTask {
   public:
      RendezvousTask() { } // empty, postpone initialization of t to start
      virtual ~RendezvousTask() {
         /* initiate termination of the thread
            associated to the Rendezvous object ... */
         rv.terminate();
         /* ... and wait for its completion */
         t.join();
      }

      virtual void operator()() = 0;

      // ...

   protected:
      // to be invoked by the most-derived constructor
      void start() {
         t = std::thread(Thread(*this));
      }
      Rendezvous<Request, Entry> rv;

   private:
      // associated thread
      std::thread t;

      // used for the synchronized termination
      std::mutex mutex;
      std::condition_variable terminating;
};
```

rendezvous.hpp

```
class Thread {
   public:
      Thread(RendezvousTask& _rt) : rt(_rt) {
      }
      void operator()() {
         try {
            rt();
         } catch (TerminationException& e) {
            /* ok, simply return */
         }
      }
   protected:
      RendezvousTask& rt;
};
```

- Ein Objekt dieser Klasse wird an *std*::*thread* übergeben.
- Aufgerufen wird die polymorphe Funktionsmethode.
- Wenn die Terminierung als Ausnahme eintrifft, wird mit *rt.terminating* signalisiert, dass der Thread abgebaut wurde.

rendezvous.hpp

```
virtual ~RendezvousTask() {
   /* initiate termination of the thread
      associated to the Rendezvous object ... */
   rv.terminate();
   /* ... and wait for its completion */
   t.join();
}
```

- Wenn das Objekt, das mit dem Thread zusammenhängt, terminiert, dann wird dieser Dekonstruktor aufgerufen.
- Dieser initiiert zuerst die Terminierung des Rendezvous-Objekts...

rendezvous.hpp

```
void terminate() {
   terminating = true;
   submitted.notify_all();
}
```

- Die *terminate*-Methode des Rendezvous-Objekts merkt sich, dass die Terminierung anläuft und weckt alle wartenden Threads auf.

rendezvous.hpp

```
rv.submitted.wait(lock);
if (rv.terminating) throw TerminationException();
```

- An den einzelnen Stellen, wo auf das Eintreffen einer Anfrage gewartet wird, muss überprüft werden, ob eine Terminierung vorliegt.
- Falls ja, muss die entsprechende Ausnahme initiiert werden.
- Diese führt dann zum Abbau des gesamten Threads der betreffenden Tasks, wobei implizit auch alle Locks freigegeben werden.