Rendezvous

- 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;
```

Umsetzung von Rendezvous in C++

- 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 erstere 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 boost::condition_variable()) {
   };
   Member(const Member& other) :
      entry(other.entry), req(other.req), done(other.done) {};
   Entry entry;
   Request req;
   boost::shared_ptr<boost::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 *boost*::*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

```
boost::mutex mutex;
boost::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
         boost::unique_lock<boost::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) {
      }
      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(EntrySet entries) throw(TerminationException) {
   boost::unique_lock<boost::mutex> lock(mutex);
   for(;;) {
      for (typename EntrySet::iterator it = entries.begin();
            it != entries.end(); ++it) {
         if (requests.find(*it) != requests.end()) {
            return *it;
         }
      }
      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.

rv-ringbuffer.hpp

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

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

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

- Der für den Ringpuffer zuständige Thread wird von der Methode *body* repräsentiert.
- Diese wird von der Template-Klasse *RendezvousTask* aufgerufen, die den Thread startet.

Auswahl der nächsten Anfrage

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;
      boost::unique_lock<boost::mutex> lock;
};
```

Abbau eines Threads

- Grundsätzlich ist der Abbau von Objekten mit zugehörigen Threads nicht-trivial.
- Die Boost-Threads-Bibliothek besteht darauf, dass
  - jedes *boost*::*mutex*-Objekt beim Abbau ungelockt sein muss und
  - auf eine *boost*::*condition_variable* niemand wartet.
- 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.

Ausnahme für die Terminierung

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.

Basisklasse für Rendezvous-Threads

rendezvous.hpp

```
template<typename Request, typename Entry>
class RendezvousTask {
   public:
      RendezvousTask() : t(Thread(*this)) {
      }
      virtual ~RendezvousTask() {
         /* initiate termination of the thread
            associated to the Rendezvous object ... */
         rv.terminate();
         /* ... and wait for its completion */
         boost::unique_lock<boost::mutex> lock(mutex);
         terminating.wait(lock);
      }

      virtual void body() = 0;

      // ...

   public:
      Rendezvous<Request, Entry> rv;

   private:
      // associated thread
      boost::thread t;

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

rendezvous.hpp

```
class Thread {
   public:
      Thread(RendezvousTask& _rt) : rt(_rt) {
      }
      void operator()() {
         try {
            rt.body();
         } catch (TerminationException& e) {
            rt.terminating.notify_all();
         }
      }
   protected:
      RendezvousTask& rt;
};
```

- Ein Objekt dieser Klasse wird an *boost*::*thread* übergeben.
- Aufgerufen wird die polymorphe *body*-Methode.
- 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 */
   boost::unique_lock<boost::mutex> lock(mutex);
   terminating.wait(lock);
}
```

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

Ablauf der Terminierung

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.

Ablauf der Terminierung

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.