13 Threads

Dieses Kapitel behandelt die folgenden Themen:

- Programmierung paralleler Abläufe mit Threads
- Synchronisation
- Thread-Steuerung: pausieren, fortsetzen, beenden
- Interrupt
- Warten auf Ereignisse/Producer-Consumer-Problem
- Monitor-Konzept
- Gleichzeitige lesende Zugriffe zulassen/Reader-Writer-Problem
- Thread-Sicherheit

Im Betriebssystem laufen verschiedene parallele Programme, auch Prozesse genannt, die jeweils einen eigenen Adressbereich haben. Zum Beispiel können ein Office-Programm und der Internet-Browser nebeneinander laufen. Die zeitliche Ausführung eines Programms kann mit einem Faden (englisch *thread*) symbolisiert werden. Dieser Faden beginnt mit dem Aufruf von `main()` und endet mit dem Ablauf von `main()`. *Innerhalb* eines solchen Programms, das selbst einen Thread darstellt, kann ein weiterer Thread gestartet werden. Das bedeutet, dass ab diesem Zeitpunkt ein weiterer Faden der Programmausführung neben dem erzeugenden Faden herläuft. Im Unterschied zu Prozessen teilen sich Threads einen Adressraum und heißen deswegen auch leichtgewichtige Prozesse. Zu einem Prozess gehörende Threads können also auf dieselben Variablen zugreifen.

Wenn es gleichviele oder mehr CPUs bzw. CPU-Kerne als parallele Prozesse/Threads gibt, können die Prozesse/Threads echt parallel abgearbeitet werden. Andernfall spricht man von *quasi-parallel*. Diese nur scheinbare Parallelität wird bei wenigen oder nur einer CPU erzeugt, indem in kurzen Zeitabständen jeder Thread mal zur Ausführung kommt.

Das Betriebssystem übernimmt die Aufteilung der Prozesse auf die CPUs. Dieser Vorgang heißt Einplanung (englisch *scheduling*).

Der Laufzeitaufwand zur Verwaltung von Threads ist geringer als der für Prozesse. Wenn ein Prozess der CPU zur Ausführung zugeteilt wird, muss vom Adressraum des vorhergehenden auf den des nun auszuführenden Prozesses umgeschaltet werden (Context-Switch). Weil Threads denselben Adressraum nutzen, ist dieser Vorgang viel weniger aufwendig. Threads sind immer dann sinnvoll, wenn verschiedene Dinge gleichzeitig getan werden sollen, zum Beispiel Bearbeiten von Dialogeingaben und gleichzeitig Laden eines Bildes. Damit wird verhindert, dass der längliche Vorgang des Bildladens die grafische Benutzungsschnittstelle blockiert. Threads werden von den Programmiersprachen auf verschiedene Art realisiert. Das Zustandsdiagramm 13.1 zeigt die wesentlichen Thread-Zustände. Die Zustände sind im Einzelnen:

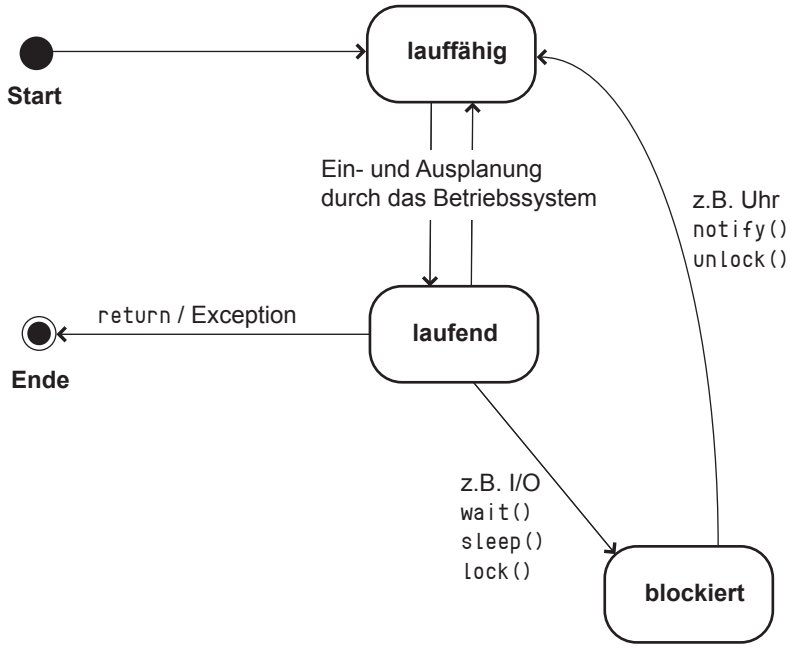

Abbildung 13.1: Thread-Zustände

- *Start*: Dies ist der Zustand nach Ablauf des Konstruktors des zugeordneten Thread-Objekts. In Java muss ein Thread erst mit dem Befehl `start()` gestartet werden, in C++ wird dies schon durch den genannten Konstruktor erledigt. Sofort danach beginnt der Thread anscheinend zu laufen. Tatsächlich wird er als lauffähig betrachtet und konkurriert mit anderen Threads und Prozessen um die Ressource CPU. Welcher Thread wann wie viel CPU-Zeit zugeteilt bekommt, bestimmt das Betriebssystem.
- *lauffähig*: In diesem Zustand kann dem Thread vom Betriebssystem CPU-Zeit zugeteilt werden.

- *laufend*: In diesem Zustand erfüllt der Thread seine Aufgabe. Da es noch viele andere Threads geben kann, wird er vom Betriebssystem immer mal wieder in den Zustand *lauffähig* versetzt, damit ein anderer Thread die CPU nutzen kann.
- *blockiert*: Ein Thread geht in diesen Zustand, wenn er warten muss. Gründe dafür können sein: Warten auf Abschluss des Datentransfers von oder zu einer Festplatte, mit `wait()` erzwungenes Warten auf eine Ressource, mit `sleep()` erzwungenes Warten für eine bestimmte Zeitdauer oder bis zu einem bestimmten Zeitpunkt.
- *Ende*: Der vom Thread zu erledigende Funktionsaufruf ist beendet.

Das folgende Beispiel zeigt drei Objekte der Klasse `thread`, denen dieselbe Funktion `f(int t)` zur Ausführung übergeben wird. Die Funktion tut nichts außer `t` Sekunden zu warten und dann die Beendigung anzuzeigen.

Hinweis

Die in [ISOC++] definierte Klasse `thread` ist zurzeit der Drucklegung dieses Buchs noch nicht hinreichend in manchen C++-Compilern verfügbar bzw. lauffähig. Aus diesem Grund wird nachfolgend die Boost Thread-Library verwendet. Die Boost-Libraries sind in vielen Fällen Vorbild für Entwicklungen des C++-Standards und kommen deshalb [ISOC++] recht nahe. Auch Qt, ein Produkt zur GUI-Entwicklung, bietet Threads (siehe Seite 469).

Listing 13.1: Thread-Beispiel

```
// cppbuch/k13/erstesBeispiel.cpp
#include <boost/thread.hpp>
#include <iostream>
using namespace std;

void f(int t) {
   // sleep: t Sekunden warten
   boost::this_thread::sleep(boost::posix_time::seconds(t));
   cout << "Thread " << boost::this_thread::get_id()
        << " : Funktion beendet! Laufzeit = " << t << " s" << endl;
}

int main() {
   boost::thread t1(f, 4);
   boost::thread t2(f, 6);
   boost::thread t3(f, 2);
   cout << "t1.get_id(): " << t1.get_id() << endl;
   cout << "t2.get_id(): " << t2.get_id() << endl;
   cout << "t3.get_id(): " << t3.get_id() << endl;
   t1.join(); // warten auf Beendigung
   t2.join();
   t3.join();
   cout << "t1.get_id(): " << t1.get_id() << endl;
   cout << "main() ist beendet" << endl;
}
```

Die Ausgabe des Programms ist

t1.get_id(): 0x8051008
t2.get_id(): 0x8051180
t3.get_id(): 0x80512f8
Thread 0x80512f8 : Funktion beendet! Laufzeit = 2 s
Thread 0x8051008 : Funktion beendet! Laufzeit = 4 s
Thread 0x8051180 : Funktion beendet! Laufzeit = 6 s
t1.get_id(): {Not-any-thread}
main() ist beendet

Was geschieht im Einzelnen?

- Zuerst werden drei Thread-Objekte erzeugt. Dem Konstruktor werden als Erstes die auszuführende Funktion (oder ein Funktionsobjekt) übergeben und als Nächstes alle Parameter, die diese Funktion benötigt. Daran kann man erkennen, dass der Thread-Konstruktor ein Template mit variabler Parameterzahl ist, wie in Abschnitt 6.5 beschrieben.
- Für jedes `thread`-Objekt wird sofort ein Thread gestartet, indem die übergebene Funktion ausgeführt wird. Im Fall eines übergebenen Funktionsobjekts wird dessen `operator()()` ausgeführt. Es ist zwischen den Threads selbst und den `thread`-Objekten, durch deren Konstruktor die Threads gestartet werden, zu unterscheiden. Der Haupt-Thread (das `main()`-Programm) läuft nach dem Start der Threads weiter, sodass es vier parallele Ausführungsfäden gibt. Jedem `thread`-Objekt ist eine Kennung zugeordnet, die von `main()` ausgegeben wird, während die anderen Threads beschäftigt sind.
- Anschließend wartet `main()` mit `t1.join()` darauf, dass sich der `t1` zugeordnete Thread beendet.
- Währenddessen meldet sich der Thread mit der kürzesten Laufzeit, dass er sich beendet hat. Innerhalb der Funktion ist das Thread-Objekt, das diese Funktion ausführt, nicht bekannt. Mit `this_thread::get_id()` kann aber die Kennung des `thread`-Objekts, das dem *aktuell* laufenden Thread zugeordnet ist, abgefragt werden – und das ist eben der, der gerade dabei ist, diese Funktion auszuführen. Das `thread`-Objekt existiert weiter und ist noch dem Thread zugeordnet, auch wenn der sich beendet hat.
- Der Thread mit der Laufzeit 4 s meldet seine Beendigung. Das ist aber gerade der Thread, auf den `main()` mit `t1.join()` wartet. Das Wort »join« bedeutet zusammenführen. Die Threads laufen ab diesem Punkt nicht mehr nebeneinander her, das heißt, der `t1` zugeordnete Thread hat aufgehört, als eigener Thread zu existieren. Das Objekt `t1` geht in den Zustand *repräsentiert keinen Thread* (mehr dazu auf Seite 424).
- Anschließend wartet `main()` mit mit `t2.join()` auf den zugeordneten Thread. Da dieser bereits fertig ist, gibt es keine Wartezeit.
- Anschließend wartet `main()` mit `t3.join()` auf den zugeordneten Thread. Dieser ist nach weiteren 2 s beendet, wie die Bildschirmausgabe zeigt, und `t3.join()` wird wirksam. Ab diesem Zeitpunkt gibt es nur noch den `main()`-Thread.
- Anschließend wird nur für `t1`, weil es für `t2` und `t3` dasselbe ergäbe, die Kennung ausgegeben. »Not-any-thread« besagt, dass `t1` zwar noch als Objekt existiert, aber keinen Thread mehr repräsentiert.

Tipp

Der Haupt-Thread `main()` sollte stets mit `join()` auf die von ihm angelegten Threads warten. Andernfalls werden diese Threads sofort bei Erreichen des Endes von `main()` zwangsweise beendet!

Das heißt, dass die Threads ihre Aufgabe nicht mehr zu Ende führen können und auch, dass darauf geachtet werden muss, dass jeder Thread wirklich terminiert, damit `main()` nicht hängen bleibt.

13.1 Die Klasse thread

Nach [ISOC++] ist die Klasse `thread` wie folgt deklariert (Auszug):

Listing 13.2: thread API nach [ISOC++] (kommentierter Auszug)

```
class thread {
public:
   class id;  // Vorwärtsdeklaration für die Identifier-Klasse
              // Ausgabe eines id-Objekts ergibt dessen Kennung (s.o.)
   thread();  // Konstruktor für einen Thread, der sich von Anfang an im Zustand
              // repräsentiert keinen Thread befindet.
   template <class F, class ...Args> explicit
   thread(F&& f, Args&&... args); // Konstruktor für eine Funktion oder einen Funktor.
   ~thread(); // Destruktor; darf nicht erreicht werden, wenn der Thread noch
              // joinable() ist.
   void swap(thread&);
   bool joinable() const; // gibt (Zustand ≠ repräsentiert keinen Thread) zurück.
   void join();  // kehrt erst nach Ende der abzuarbeitenden Funktion zurück und
                 // setzt den Zustand auf repräsentiert keinen Thread.
   void detach(); // löst die Verbindung zum Thread, setzt den Zustand auf repräsentiert
                  //  keinen Thread und kehrt sofort zurück (Beispiel siehe unten).
   id get_id() const; // gibt Identitäts-Objekt des Threads zurück bzw. id(),
                      // falls der Thread im Zustand repräsentiert keinen Thread ist.
   thread(const thread&) = delete;         // Kopierkonstruktor verbieten
   thread& operator=(const thread&) = delete; // Zuweisung verbieten
};
```

Die ausführliche Erklärung für `&&` lesen Sie erst in Abschnitt 22.2. Gemeint ist, dass Links-Werte in den Konstruktor hineinkopiert werden (also normale Übergabe per Wert), dass aber temporäre Objekte hinein-*bewegt* werden (Wegoptimieren des Kopiervorgangs). Für den Ablauf im Thread macht es keinen Unterschied. Es wird intern mit einer Kopie der Parameter gearbeitet, egal wie optimiert die Inhalte der Parameter zustande gekommen sind. Die Semantik ist genau so, als ob dort

```
template <class F, class ...Args> thread(F f, Args... args);
```

stünde. Mit `= delete` wird dem Compiler mitgeteilt, dass er auf die automatische Erzeugung verzichten soll. Ein weiteres wichtiges API ist im Namespace `this_thread`. Dieser Namespace enthält Funktionen, die sich auf den aktuell laufenden Thread beziehen:

Listing 13.3: this_thread

```
namespace this_thread {
   thread::id get_id();  // gibt das Id-Objekt des aktuellen Threads zurück,
                         // Verwendung im Beispiel oben.
   void yield();         // Mitteilung an das Betriebssystem, dass dieser
                         // Thread nicht eilig ist und andere vorgezogen werden können.
   // Thread für eine bestimmte Zeitdauer schlafen legen:
   template <class Rep, class Period>
   void sleep_for(const chrono::duration<Rep, Period>& rel_time);
   // Thread bis zu einem bestimmten Zeitpunkt schlafen legen:
   template <class Clock, class Duration>
   void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
}
```

`<chrono>` ist in Abschnitt 27.7 ab Seite 760 beschrieben. Das in den Beispielen benutzte Boost-API für `sleep()` (statt `sleep_for` und `sleep_until` gemäß [ISOC++]) ist:

- Thread für eine bestimmte Zeitdauer schlafen legen:

  ```
  // Namespace boost::this_thread-Funktion:
  template<typename TimeDuration>
  void sleep(TimeDuration const& rel_time);
  ```

  Ein direkt umsetzbares Anwendungsbeispiel:

  ```
  boost::this_thread::sleep(boost::posix_time::seconds(anzahl));
  ```

  `seconds` kann je nach Anwendungszweck durch `hours`, `minutes`, `millisec` und `nanosec` ersetzt werden. Auch Kombinationen sind möglich, etwa

  ```
  using namespace boost::posix_time;
  time_duration dauer = hours(5) + minutes(3) + seconds(59);
  boost::this_thread::sleep(dauer);
  ```

- Thread bis zu einem bestimmten Zeitpunkt schlafen legen:

  ```
  // static Elementfunktion der Klasse thread:
  void sleep(system_time const& abs_time);
  ```

  Ein Beispiel, um fünf Minuten zu warten, bis das Ei weich gekocht ist (Auszug):

  ```
  boost::system_time jetzt = boost::get_system_time();
  boost::thread::sleep(jetzt + boost::posix_time::minutes(5));
  ```

Zustände von thread-Objekten

`thread`-Objekte unterscheiden sich von den zugeordneten Threads. Die Zustände von `thread`-Objekten und Threads sind daher ganz verschieden. `thread`-Objekte können sich in zwei Zuständen befinden:

- *repräsentiert einen Thread*: In diesem Zustand ist das Thread-Objekt einem Thread, der von ihm gestartet wurde, zugeordnet. Die Methode `get_id()` gibt das zugehörige Identitäts-Objekt zurück.

- *repräsentiert keinen Thread*: Dieser Zustand bedeutet, dass das Thread-Objekt keinem Thread (mehr) zugeordnet ist. Die Methode `get_id()` gibt ein vom Standardkonstruktor der Klasse `id` erzeugtes Objekt zurück. Das heißt aber nicht unbedingt, dass der Thread als Ausführungsfaden nicht mehr existiert, sondern nur, dass er durch das Thread-Objekt nicht mehr repräsentiert wird – das ist nicht dasselbe! Der Zustand wird durch den Aufruf einer der Methoden `join()` oder `detach()` erreicht. Das Verb »to detach« bedeutet »(Verbindung) lösen«.
  - `t.join()` blockiert den Aufrufer, bis der Thread `t` beendet ist. Nach dem Aufruf ist `t` im Zustand *repräsentiert keinen Thread*, der Aufrufer setzt seinen Ablauf fort.
  - `t.detach()` blockiert den Aufrufer *nicht*: Der Aufruf kommt sofort zurück mit dem Ergebnis, dass `t` im Zustand *repräsentiert keinen Thread* ist. Der ehemals zugeordnete Thread läuft währenddessen weiter, bis er beendet ist. Das folgende kleine Programm zeigt den Effekt:

Listing 13.4: detach()

```
// Auszug aus cppbuch/k13/detach.cpp
// Rest wie oben, siehe cppbuch/k13/erstesBeispiel.cpp
int main() {
   boost::thread t1(f, 4);
   boost::thread t2(f, 6);
   boost::thread t3(f, 2);
   cout << "t1.get_id(): " << t1.get_id() << endl;
   cout << "t2.get_id(): " << t2.get_id() << endl;
   cout << "t3.get_id(): " << t3.get_id() << endl;

   t1.detach();
   cout << "t1.detach(): " << t1.get_id() << endl;
   t3.detach();
   cout << "t3.detach(): " << t3.get_id() << endl;
   t2.join(); // warten auf Beendigung des längstdauernden Threads
   cout << "main() ist beendet" << endl;
}
```

Die Ausgabe des Programms ist:

```
t1.get_id(): 0x8051008
t2.get_id(): 0x8051180
t3.get_id(): 0x8051318
t1.detach(): {Not-any-thread}
t3.detach(): {Not-any-thread}
Thread 0x8051318 : Funktion beendet! Laufzeit = 2 s
Thread 0x8051008 : Funktion beendet! Laufzeit = 4 s
Thread 0x8051180 : Funktion beendet! Laufzeit = 6 s
main() ist beendet
```

13.2 Synchronisation

Programme, die »normalerweise« korrekt funktionieren, tun dies nicht unbedingt, wenn Threads im Spiel sind. Dazu ein kleines Beispiel: Mit der Klasse `Gerade` kann eine Folge gerader Zahlen erzeugt werden. Der Aufruf von `next()` liefert die jeweils nächste Zahl:

Listing 13.5: Klasse für gerade Zahlen – nicht threadsafe

```
1  class Gerade {
2  public:
3     Gerade()
4        : n(0) {
5     }
6     int next() {
7        ++n;
8        ++n;
9        return n;
10    }
11 private:
12    int n;
13 };
```

Eine Testfunktion könnte wie folgt aussehen:

```
Gerade g;

void testeGerade() {
   for(int i=0; i < 10000; ++i) {
      int wert = g.next();
      if(wert % 2 != 0) {
         cout << wert << " ist ungerade!"<< endl;
         break;
      }
   }
}
```

Im sequentiellen Fall ist garantiert, dass `next()` eine gerade Zahl liefert. Wenn jedoch zwei oder mehrere Threads auf dasselbe Objekt zugreifen, kann es Fehler geben. Wenn nur zwei Threads A und B existieren und `n` den Anfangswert 0 hat, kann es zu folgender Abfolge kommen:

1. Thread A führt `next()` bis Zeile 7 einschließlich aus und wird dann vom Betriebssystem in die Warteschlange wartender Threads gepackt, damit Thread B drankommt. Das Attribut `n` hat den Wert 1.
2. Thread B führt `next()` aus und erhöht `n` zweimal. Es wird 3 zurückgegeben – Fehler!
3. Thread A kommt wieder dran und führt `next()` zu Ende aus; es wird 4 zurückgegeben.

Das Problem lässt sich leicht visualisieren, wenn man mehrere Threads auf das obige Objekt `g` loslässt. Am einfachsten lässt sich das durch eine Gruppe von Threads realisieren:

```
// Auszug aus cppbuch/k13/gerade.cpp
int main() {
  boost::thread_group threads; // mehr zu thread_group auf Seite 428
```

```
   for (int i = 0; i < 20; ++i) {
      threads.create_thread(testeGerade);
   }
   threads.join_all();
}
```

Das Programm wird mal ungerade Zahlen ausgeben, mal auch nicht – abhängig von der Einplanung der Threads. Der Effekt, dass zufällig mal der eine und mal der andere Thread zuerst die Daten ändert, heißt *data race*. Eine Bedingung, die dazu führt, nennt man *race condition*. Die Ursache des Problems liegt darin, dass eine korrekte Abarbeitung der Funktion `next()` *atomar* (unteilbar) sein muss. Ein Thread sollte nur dann `next()` betreten dürfen, wenn jeder andere Thread die Funktion verlassen hat. Bereiche, in denen Daten verändert werden und die dem Zugriff mehrerer Threads ausgesetzt sind, wie hier die Zeilen 7 und 8 der Klasse, heißen *kritischer Bereich* (englisch *critical section*). Der Zugriff auf einen kritischen Bereich muss *synchronisiert* werden, indem sich die Zugriffe gegenseitig ausschließen (englisch *mutual exclusion*). Die Klasse `mutex` steuert den gegenseitigen Ausschluss. Der Ablauf wäre etwa wie folgt:

```
mutex mtx;

int next() {       // erster Ansatz
   mtx.lock(); // Zugriff für alle anderen verbieten
   // Anfang des kritischen Bereichs
   ++n;
   ++n;
   // Ende des kritischen Bereichs?
   mtx.unlock(); // Zugriff wieder freigeben. Immer noch problematisch, siehe Text
   return n;
}
```

Diese Idee hat zwei Schwächen:

- `mtx.unlock()` ist nicht die letzte Anweisung. Daher kann ein anderer Thread nach `mtx.unlock()`, aber vor `return`, `next()` ausführen und das Ergebnis verfälschen. Eine Lösung wäre natürlich, in der Funktion auf `mtx` zu verzichten und jeden Aufruf von `next()` einzuschachteln, etwa

  ```
  mtx.lock();
  int wert = g.next();
  mtx.unlock();
  // wert weiterverarbeiten
  ```

  Diese richtige Lösung ist fehleranfällig, weil das Paar der `lock()`/`unlock()`-Aufrufe doch mal vergessen wird, und sie ist auch sehr umständlich.
- Es kann vorkommen, dass in einem kritischen Bereich eine Exception geworfen wird. Die Folge ist, dass `mtx.unlock()` nicht aufgerufen und nichts freigegeben wird. Jeder nachfolgende Thread bleibt danach bei `mtx.lock()` stehen – das Programm hängt.

Deshalb wählt man einen anderen, komfortableren und sichereren Weg, indem ein Lock-Objekt angelegt wird:

```
mutex mtx;
// ...
```

```
int next() {          // richtiger Ansatz:
   Lock lock(mtx);    // Lock-Objekt anlegen
   // Anfang des kritischen Bereichs
   ++n;
   ++n;
   return n;  // Ende des kritischen Bereichs
}
```

Der `lock`-Konstruktor ruft `mtx.lock()` und realisiert damit die Sperre. Der Destruktor ruft `mtx.unlock()`, aber erst beim Verlassen des Funktionsbereichs, also bei der schließenden Klammer des Funktionsblocks. Dieser Ansatz nennt sich englisch *scoped locking* oder *guard* ([SSRB]). Dahinter verbirgt sich nichts anderes als das Prinzip »Resource Acquisition Is Initialization« (siehe RAII im Glossar). C++ verwendet für den Zweck des *scoped locking* die Klassen `lock_guard` oder `unique_lock`. Letztere hat ein paar mehr Möglichkeiten wie etwa `lock()` und `unlock()`; Funktionen, die `lock_guard` nicht hat und die in diesem Kontext nicht gebraucht werden. Die obige Klasse `Gerade` wird wie folgt ergänzt, damit jeder Zugriff auf `next()` garantiert eine gerade Zahl liefert:

Listing 13.6: Klasse für gerade Zahlen – threadsafe

```
// Auszug aus cppbuch/k13/gerade.cpp
namespace {
   boost::mutex mtx;
}

class Gerade {
public:
   Gerade() : n(0) {
   }
   int next() {
      boost::lock_guard<boost::mutex> lock(mtx); // neu!
      ++n;
      ++n;
      return n;
   }
private:
   int n;
};
```

13.2.1 Thread-Group

Eine Thread-Group fasst mehrere Threads zusammen, um Funktionen für alle Threads der Gruppe mit nur einer Anweisung realisieren zu können. In [ISOC++] ist eine Thread-Group nicht vorgesehen, es gibt sie aber in der Boost Library. Die Schnittstelle:

Listing 13.7: Boost thread_group

```
class thread_group {
public:
    thread_group();                      // Konstruktor
    ~thread_group();                     // Destruktor
    template<typename F>
    thread* create_thread(F threadfunc); // Thread erzeugen und starten
```

```
    void add_thread(thread* thrd);     // Thread hinzufügen
    void remove_thread(thread* thrd); // Thread entfernen
    void join_all();                   // join() für alle ausführen
    void interrupt_all();              // interrupt() für alle ausführen
    int size() const;                  // gibt Anzahl der Threads zurück
};
```

Der Destruktor führt `delete` auf alle erzeugten und hinzugefügten Threads aus, woraus sich ergibt, dass die Threads von `create_thread()` mit `new` angelegt werden. Auch darf `add_thread()` nur ein Zeiger auf ein mit `new` erzeugtes Objekt übergeben werden, damit der Destruktor nicht ein Objekt zu löschen versucht, das auf dem Heap nicht existiert. Wie funktioniert eine Thread-Group? Am einfachsten ist es, sich einen Vektor vorzustellen, in dem Zeiger auf Threads gespeichert werden, etwa wie hier gezeigt:

Listing 13.8: Mögliche ThreadGroup-Implementierung

```
class ThreadGroup { // unvollständig
public:
   ~ThreadGroup() {
      for(size_t i = 0; i < pthreads.size(); ++i)
         delete pthreads[i];
   }
   template<typename T>
   boost::thread* create_thread(T t) {
      pthreads.push_back(new boost::thread(t)); // Thread speichern und starten
      return pthreads[pthreads.size()-1];
   }
   void join_all() {
      for(size_t i = 0; i < pthreads.size(); ++i)
         pthreads[i]->join();
   }
   // weitere Methoden
private:
   std::vector<boost::thread*> pthreads;
};
```

13.3 Thread-Steuerung: pausieren, fortsetzen, beenden

In diesem Abschnitt geht es darum, wie ein Thread kontrolliert angehalten, fortgesetzt und schließlich beendet werden kann. Dazu muss erst ein kleines Problem gelöst werden, weil die Klasse `thread` aus gutem Grund dafür keine Methoden hat. `thread` kann zum Beispiel nicht wissen, ob vor dem Pausieren bestimmte Arbeiten noch erledigt werden müssen. Ein anderes Problem besteht in der Art, wie die Übergabe des Funktionsobjekts in der Klasse `thread` vorgesehen ist. Sehen Sie sich das folgende Beispiel an:

```
class Funktor {
public:
   void operator()() {
      // eigentliche Arbeit erledigen
   }
   void pause() { ... }
   void fortsetzen() { ... }
};

int main() {
   Funktor f;
   thread t(f);  // Ab hier wird operator()() ausgeführt.
   // ... warten
   f.pause();    // wirkt nicht!
}
```

Dass der Aufruf `f.pause()` nicht wirkt, liegt daran, dass `f` per Wert übergeben wird. Innerhalb des Threads wird `operator()()` für eine lokale Kopie von `f` ausgeführt. Diese lokale Kopie ist von `main()` aus nicht erreichbar. Deshalb gibt es eine Hüllklasse (englisch *wrapper*) `CallWrapper`, die eine Referenz auf den Funktor enthält. Darüber sind alle Methoden des Funktors erreichbar. Die Klasse, die die eigentliche Arbeit erledigt, heißt `Worker`. Zunächst stelle ich die Klasse `CallWrapper` und die Anwendung vor, damit Sie sich ein Bild vom Ablauf bilden können, der einen Arbeitstag simuliert. Eine Sekunde entspricht einer simulierten Stunde.

```
// Auszug aus cppbuch/k13/flags.cpp
template<class Callee> // Callee = Klasse des aufzurufenden Funktors
class CallWrapper {
public:
   CallWrapper(Callee& c) : callee(c) {}
   void operator()() {
      callee();                    // = callee.operator()()
   }
private:
   Callee& callee;
};
```

Hinweis

C++ stellt die Klasse `reference_wrapper` zur Verfügung, die dasselbe wie die Klasse `CallWrapper` leistet (und noch ein wenig mehr), siehe Seite 761. Aus diesem Grund wird nur im direkt folgenden Beispiel zur Demonstration `CallWrapper` verwendet, danach ausschließlich die Funktion `ref()`, die ein `reference_wrapper`-Objekt zurückgibt.

Weil das `Worker`-Objekt bei Instanziierung des Templates mit der Klasse `Worker` per Referenz übergeben wird, bewirkt der Aufruf von `operator()()` stets, dass `operator()()` für das übergebene Originalobjekt `c` aufgerufen wird, auch wenn das `CallWrapper`-Objekt selbst eine Kopie ist – der Wert des Attributs (Referenz `callee`) bleibt bei einer Kopie erhalten. Weiter geht es mit `main()` und der Simulation eines Arbeitstages:

Listing 13.9: Simulation eines Arbeitstages

```
// Auszug aus cppbuch/k13/workerthread/main.cpp
int main() {
  Worker worker;
  CallWrapper aufrufer(worker);  // Referenz auf Worker übergeben
  boost::thread t(aufrufer);  // Thread anlegen und starten
  int stunde = 8;
  while(!worker.istBeendet()) {
    boost::this_thread::sleep(boost::posix_time::seconds(1));
    cout << ++stunde << " Uhr: ";
    switch(stunde) {
    case 10 : worker.warten();       // Pause
              break;
    case 13 : worker.weiter();       // Fortsetzung
              break;
    case 16: worker.beenden();       // Ende
              t.join();              // Ende abwarten
              break;
    default: if(stunde > 16) {
               throw "Fehler!";
             }
    }
  }
}
```

Die Ausgabe des Programms ist:

Worker bei der Arbeit!
9 Uhr: Worker bei der Arbeit!
10 Uhr: Worker bei der Arbeit!
Worker pausiert!
11 Uhr: 12 Uhr: 13 Uhr: Worker macht weiter!
14 Uhr: Worker bei der Arbeit!
15 Uhr: Worker bei der Arbeit!
16 Uhr: Worker macht Feierabend!

Die Steuerung läuft über die Flags `pause` und `beendet` der Klasse `Worker`:

Listing 13.10: Klasse Worker

```
// Auszug aus cppbuch/k13/flags.cpp
class Worker {
public:
  Worker()
    : pause(false), beendet(false) {
  }
  void operator()() {
    while(!beendet) {
      cout << "Worker bei der Arbeit!" << endl;
      pauseOderNicht();
      boost::this_thread::sleep(boost::posix_time::seconds(1));
    }
    cout << "Worker macht Feierabend!" << endl;
  }
```

```
   void warten() {
      boost::lock_guard<boost::mutex> lock(mtxPause);
      pause = true;
   }
   void weiter() {
      boost::lock_guard<boost::mutex> lock(mtxPause);
      pause = false;
      cond.notify_one();
   }
   void beenden() {
      beendet = true;
      cond.notify_one(); // ... falls im Zustand pausierend
   }
   bool istBeendet() const {
      return beendet;
   }
private:
   bool pause;
   bool beendet;
   boost::mutex mtxPause;
   boost::condition_variable cond;

   void pauseOderNicht() {
      boost::mutex mtx;
      boost::unique_lock<boost::mutex> lock(mtx);
      while(pause && !beendet) {
         cout << "Worker pausiert!" << endl;
         cond.wait(lock);
         cout << "Worker macht weiter!" << endl;
      }
   }
};
```

Hinweis

Der lesende und schreibende Zugriff auf die Variable `beendet` ist nicht mit einem Lock synchronisiert, weil es nicht darauf ankommt: Die Änderung auf `true` ist irreversibel.

Im `main()`-Programm wird der Thread gestartet und jede Sekunde die Variable `stunde` um eins erhöht. Um 10 »Uhr« wird pausiert, veranlasst durch den Aufruf von `warten()`. Mit `weiter()` wird der Thread um 13 Uhr fortgesetzt und um 16 Uhr der simulierten Zeit beendet. Die Arbeit des Threads wird in der Methode `operator()()` geleistet, solange das Flag `beendet` nicht gesetzt ist. Entscheidend in der Methode `operator()()` ist der Aufruf der privaten Methode `pauseOderNicht()`. In dieser Methode wird gewartet, bis das Flag `pause`, gesetzt durch die Methode `warten()`, zurückgesetzt worden ist. Auf den ersten Blick sind zwei einfache Lösungen denkbar, die jedoch beide ihre Nachteile haben:

```
// ständige Abfrage: Tun Sie das nicht!
while(pause) {
}
```

Bei dieser Methode, englisch *busy waiting* genannt, wird sinnlos CPU-Zeit verbraten.

```
// periodische Abfrage: Tun Sie das nicht, wenn es eine Alternative mit wait gibt!
while(pause) {
   sleep(seconds(1));
}
```

Die periodische Abfrage wird englisch »Polling« genannt. Der Nachteil besteht darin, dass die eingestellte Periode zwangsweise abläuft, wenn `pause` inmitten der Sekunde zurückgesetzt wird. Alternativ `seconds(1)` durch `millisec(10)` zu ersetzen, würde den genannten Nachteil reduzieren, wäre aber durch einen erheblichen Aufwand, der durch die vielen nutzlosen `sleep()`-Aufrufe entsteht, zu bezahlen. Die richtige Lösung ist etwas komplizierter, aber dafür kostet das Warten fast nichts, weil der `wait()`-Aufruf vom Betriebssystem unterstützt wird. Der Ablauf in `pauseOderNicht()` ist im Einzelnen (`pause` ist gesetzt):

- `cond.wait(lock)` sorgt dafür, dass der Thread in eine Warteschlange des Betriebssystems versetzt wird und somit nicht mehr aktiv ist. `cond` ist eine Bedingungsvariable, die Attribut der Klasse ist. Bedingungsvariablen stehen die Funktionen `wait()` und `notify()` (in Varianten) zur Verfügung. Beim Aufruf von `wait()` wird das Lock-Objekt übergeben. Das Lock-Objekt ist zwar funktionslokal, jedoch nicht die Mutex-Variable, die übergeben wird.
- Um 13 Uhr simulierter Zeit wird in `main()`, also von einem anderen, aktiven Thread `worker.weiter()` aufgerufen. Diese Funktion setzt das Flag `pause` zurück und ruft `cond.notify_one()` auf. Der Aufruf von `notify_one()` bewirkt, dass genau ein Thread aus der Warteschlange befreit wird. In diesem Beispiel kann es auch nur einer sein. Die Funktion `wait()` kehrt zurück.
- Das Flag wird in der `while()`-Bedingung noch einmal überprüft, weil es inzwischen wieder gesetzt worden sein könnte. Wenn nicht, wird die Schleife kein zweites Mal ausgeführt, die Funktion `pauseOderNicht()` kehrt zurück, und die Schleife in `operator()()` kann weiterlaufen.
- Um 16 Uhr simulierter Zeit wird in `main()` `worker.beenden()` aufgerufen. Diese Funktion setzt das Flag `beendet`, das in der Schleifenbedingung in `operator()()` abgefragt wird. Es ist jedoch möglich, dass `beenden()` gerade dann aufgerufen wird, wenn der Thread in `pauseOderNicht()` wartet. Um den Thread wirklich zu Ende zu bringen, muss er in diesem Fall befreit werden; dies ist der Grund für die Anweisung `cond.notify_one()` in der Funktion `beenden()`. Falls ein Thread nicht in `wait()` hängt, ist der Aufruf von `notify_one()` ohne Wirkung.
- Das abschließende `t.join()` in `main()` sorgt dafür, dass `main()` wartet, bis sich der Thread wirklich beendet hat, weil der Vorgang des Beendens einige Zeit in Anspruch nehmen kann.

Statt `notify_one()` könnte auch `notify_all()` verwendet werden. `notify_all()` befreit *alle* der betreffenden Bedingung zugeordneten wartenden Threads. Von denen wählt das Betriebssystem einen aus, der zum Zuge kommen kann. Die anderen werden wieder in die Warteposition geschickt.

Mehr zum Unterschied der beiden Funktionen finden Sie auf Seite 576.

Einen Thread pro Objekt garantieren

In der obigen `main()`-Funktion wäre es möglich, dass zwei `thread`-Objekte dasselbe `Worker`-Objekt verwenden, was zu Fehlern führte. Wenn Sie garantieren wollen, dass pro `Worker`-Objekt genau ein Thread läuft, konstruieren Sie am besten eine Klasse ähnlich wie `Worker`, zum Beispiel mit Namen `WorkerThread`, die aber ein Thread-Objekt als Attribut hat. Nach Erzeugung eines `WorkerThread`-Objekts kann der Thread mit einer Methode `start()`, die ein `thread`-Objekt anlegt, gestartet werden:

```
// Auszug aus cppbuch/k13/workerthread/WorkerThread.h
void start() {
   derThread = boost::thread(std::ref(*this));
}
```

`derThread` ist ein Attribut, das anfangs mit dem `thread`-Standardkonstruktor initialisiert wird, also keinen Thread repräsentiert. In `start()` wird ihm der tatsächlich zu startende Thread zugewiesen. Die Methode `beenden()` sorgt für die `join()`-Operation, sodass sie nicht von außen aufgerufen werden muss:

```
void beenden() {
   beendet = true;
   cond.notify_one(); // ... falls im Zustand pausierend
   derThread.join();
}
```

Das vollständige Beispiel finden Sie im Verzeichnis *cppbuch/k13/workerthread*.

13.4 Interrupt

Hinweis

Einen Thread mit der Funktion `interrupt()` zu unterbrechen, ist in [ISOC++] nicht vorgesehen, im Gegensatz zu den Boost-C++-Threads. Um langfristig portabel zu bleiben, empfehle ich die daher die oben skizzierte Steuerung mit Flags. Dieser Abschnitt beschreibt das Interrupt-Konzept für Boost-C++-Threads (in Java wird es auf ähnliche Art realisiert). Abschnitt 13.5 zeigt, wie ein `wait()` ohne Interrupt unterbrochen werden kann.

Interrupt heißt auf Deutsch Unterbrechung, und der Name der Funktion `interrupt()` suggeriert, als würde von ihr ein Thread unterbrochen. Tatsächlich ist es jedoch nur eine Interrupt-Anforderung: Es wird ein internes Flag gesetzt, das von allen Varianten der `wait()`-, `join()`- und `sleep()`-Methoden ausgewertet wird. Der Interrupt-Mechanismus kann ganz abgeschaltet werden. Falls das nicht geschehen ist, werfen die genannten Methoden bei gesetztem Flag eine `boost::thread_interrupted`-Exception. Das heißt auch: Falls die Exception folgenlos abgefangen wird, läuft der Thread einfach weiter – ohne Unterbrechung. Zur Demonstration wird die obige Klasse `Worker` angepasst, indem un-

ter anderem das Attribut `beendet` gestrichen wird. Zur Abwechslung wird die Abfrage in `main()`, ob der Thread noch läuft, durch die Zustandsabfrage des `thread`-Objekts gelöst.

```
// Auszug aus cppbuch/k13/interr.cpp
int main() {
  Worker worker;
  boost::thread t(std::ref(worker)); // reference_wrapper
  int stunde = 8;
  boost::thread::id keinThread;              // neu
  while(t.get_id() != keinThread) {          // neu
    boost::this_thread::sleep(boost::posix_time::seconds(1));
    cout << ++stunde << " Uhr: ";
    switch(stunde) {
    case 10 : worker.warten();
              break;
    case 13 : worker.weiter();
              break;
    case 16: t.interrupt();          // neu
             t.join();               // wartet und ändert Zustand
             break;
    default: if(stunde > 16) {
               throw "Fehler!";
             }
    }
  }
}
```

In der Klasse `Worker` entfallen die Methoden `beenden()` und `istBeendet()`. Die Methoden `operator()()` und `pauseOderNicht()` werden leicht modifiziert:

Listing 13.11: Interrupt-Auswertung

```
// Auszug aus cppbuch/k13/interr.cpp
void operator()() {
  try {
    while(true) {              // Abbruch mit Interrupt
      cout << "Worker bei der Arbeit!" << endl;
      pauseOderNicht();
      boost::this_thread::sleep(boost::posix_time::seconds(1));
    }
  } catch(const boost::thread_interrupted& e) {
    ; // nichts machen, while-Schleife ist beendet
  }
  cout << "Worker macht Feierabend!" << endl;
}
```

Sofort nach Aufruf von `t.interrupt()` in `main()` wird eine `boost::thread_interrupted`-Exception geworfen. Damit wird die Schleife verlassen, und der Thread ist beendet. Für die Exception kann es zwei Quellen geben:

- Die Methode `sleep()` in der Funktion `operator()()` und
- Die Methode `wait()` in der Funktion `pauseOderNicht()`.

Im letzten Fall wird die während `wait()` geworfene Exception automatisch an den Aufrufer `operator()()` weitergegeben.

```
// Auszug aus cppbuch/k13/interr.cpp
void pauseOderNicht() {
   boost::mutex mtx;
   boost::unique_lock<boost::mutex> lock(mtx);
   while(pause) {
      cout << "Worker pausiert!" << endl;
      cond.wait(lock);                // kann Exception werfen
      cout << "Worker macht weiter!" << endl;
   }
}
```

13.5 Warten auf Ereignisse

Das Producer-Consumer-Problem ist eins der sehr gut bekannten Probleme der Thread-Programmierung. Es geht um Folgendes: Ein Produzent produziert etwas, legt es ab, und ein Konsument holt sich das abgelegte Objekt zur Weiterverarbeitung. Die Kapazität der Ablage ist begrenzt. Die skizzierte Aufgabenstellung findet sich in jeder Fabrik oder Fabriksimulation. Eine Maschine bearbeitet ein Werkstück und legt es auf einen Stapel oder ein Fließband, von wo es einer anderen Maschine zur Weiterverarbeitung zugeführt wird. Diese andere Maschine wartet auf das Ereignis, dass das nächste Werkstück zur Verfügung steht.

Es geht dabei im Wesentlichen um die Abstimmung von Producer und Consumer: Der Erste kann nichts ablegen, wenn die Ablage voll ist, und der Letzte kann nichts entnehmen, wenn sie leer ist. Gekoppelte Systeme dieser Art zur Fabriksimulation sind interessant, weil mit ihnen die Durchlaufzeit vom Anfang bis zum fertigen Produkt berechnet werden kann. Störungen im Ablauf oder Verbesserungen durch Einführung paralleler Strecken lassen sich gut simulieren. Um das Problem zu vereinfachen, nehme ich im Folgenden an, dass die Ablage nur Platz für ein einziges Element hat. Ferner sollen die »Elemente« durch einfache `int`-Zahlen dargestellt werden, wie in der Abbildung 13.2 gezeigt.

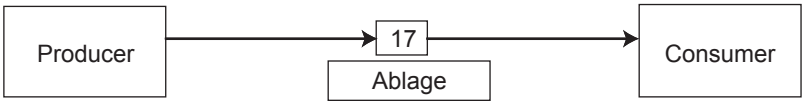

Abbildung 13.2: Producer-Consumer-Problem

Im Programmbeispiel bekommt der Producer eine Nummer `id` zur Identifizierung. Nach dem Start »produziert« er 5 `int`-Zahlen in einem zeitlich zufälligen Abstand und legt sie im `Ablage`-Objekt ab, dessen Referenz dem Konstruktor übergeben wird. Der Producer kann im Fehlerfall bei `wait()` hängenbleiben, weil der Consumer ausgefallen ist. Deshalb kann das Warten mit einer Exception, von `put()` geworfen, beendet werden.

Listing 13.12: Producer

```
// cppbuch/k13/prodcons/Producer.h
#ifndef PRODUCER_H
#define PRODUCER_H
#include <iostream>
#include<Random.h>
#include"Ablage.h"

class Producer {
public:
   Producer(Ablage& a, int i)
      : ablage(a), id(i), zufall(500) {
   }
   void operator()() {
      for(int i = 0; i < 5; ++i) {
         int wert = id*10 + i;
         boost::this_thread::sleep(
                     boost::posix_time::millisec(200 + zufall()));
         try {
            ablage.put(wert);                    // Exception-Quelle
            boost::lock_guard<boost::mutex> lock(ausgabeMutex);
            std::cout << "Producer Nr. " << id
                      << "  legt ab: " << wert << std::endl;
         } catch(...) {
             break; // Producer ist beendet
         }
      }
      boost::lock_guard<boost::mutex> lock(ausgabeMutex);
      std::cout << "Producer " << id << " beendet sich." << std::endl;
   }
private:
   Ablage& ablage;
   int id;
   Random zufall;
};
#endif
```

Falls die Ablage belegt ist, muss der Producer bei `put()` warten, bis sie frei ist. Die Ausgabeanweisungen sind nicht threadsicher; es kann zu einem *data race* kommen. `ausgabeMutex` sorgt deshalb dafür, dass sich die Nachrichten von Producer und Consumer auf dem Bildschirm nicht vermischen. `ausgabeMutex` ist in der unten gezeigten inkludierten Datei *Ablage.h* enthalten. Der Destruktor des Locks gibt die Mutex-Variable automatisch frei. Falls nach der Ausgabe noch langandauernde Tätigkeiten stattfinden (hier nicht der Fall), kann man die Freigabe durch Einschließen des Locks und der Ausgabe in einen Block beschleunigen:

```
{  // Block-Beginn
   boost::lock_guard<boost::mutex> lock(ausgabeMutex);
   std::cout << "Ausgabe" << std::endl;
}  // Block-Ende: Freigabe von ausgabeMutex durch Destruktor
// ... weitere Anweisungen
```

Der Consumer holt sich einen Wert aus der Ablage. Ist sie leer, muss der Consumer bei `get()` warten. Der Producer beendet sich nach Erledigung seiner Aufgabe automatisch; der Consumer hingegen kann nicht wissen, wann er den letzten Wert verarbeitet hat, und wartet vielleicht vergeblich. Aus diesem Grund ist vorgesehen, dass das Warten mit einer Exception, von `get()` geworfen, beendet wird.

Listing 13.13: Consumer

```
// cppbuch/k13/prodcons/Consumer.h
#ifndef CONSUMER_H
#define CONSUMER_H
#include <iostream>
#include"Ablage.h"

class Consumer {
public:
   Consumer(Ablage& a)
      : ablage(a) {
   }

   void operator()() {
      try {
         while(true) {                          // Abbruch mit Exception
            int wert = ablage.get();            // Exception-Quelle
            boost::lock_guard<boost::mutex> lock(ausgabeMutex);
            std::cout << "Consumer hat "
                      << wert << " geholt." << std::endl;
         }
      } catch(...) {
         boost::lock_guard<boost::mutex> lock(ausgabeMutex);
         std::cout << "Consumer beendet sich." << std::endl;
      }
   }
private:
   Ablage& ablage;
};
#endif
```

Die Synchronisation wird durch das `Ablage`-Objekt, das eine gemeinsam genutzte Ressource darstellt, vorgenommen. Im zugehörigen `main()`-Programm gibt es zwei Producer und einen Consumer:

Listing 13.14: Producer-Consumer-Beispiel

```
// cppbuch/k13/prodcons/main.cpp
#include <boost/thread.hpp>
#include"Ablage.h"
#include"Producer.h"
#include"Consumer.h"
using namespace std;

int main() {
   Ablage ablage;
   Producer p1(ablage, 1);
```

```
   Producer p2(ablage, 2);
   Consumer c(ablage);
   // Threads starten
   boost::thread tp1(p1);
   boost::thread tp2(p2);
   boost::thread tc(c);
   // Ende der Producer abwarten
   tp1.join();
   tp2.join();
   // warten, bis alles abgeholt ist
   boost::this_thread::sleep(boost::posix_time::seconds(1));
   ablage.beenden();   // wartende Prozesse beenden
   tc.join();          // Ende des Consumers abwarten
}
```

Monitor-Konzept

Diese Klasse von Problemen zeichnet sich dadurch aus, dass viele Threads gleichzeitig auf Methoden eines Objekts, die kritische Bereiche lesen oder ändern, zugreifen können. Die Lösung ist eine Synchronisation, die es nur einem einzigen Thread zurzeit erlaubt, eine Methode aufzurufen. Man sagt, dass die Methoden *synchronisiert* sind. Ein Objekt, dass diese Voraussetzung erfüllt, heißt *Monitor*-Objekt. »Monitor« ist ein aus der Betriebssystemprogrammierung stammender Begriff. Diese Voraussetzung kann leicht erfüllt werden, wenn es ein spezielles Mutex-Attribut gibt, für das am Eingang jeder kritischen Methode ein Lock-Objekt angelegt wird, wie die Klasse `Ablage` zeigt:

Listing 13.15: Gemeinsame Ressource: Ablage

```
// cppbuch/k13/prodcons/Ablage.h
#ifndef ABLAGE_H
#define ABLAGE_H
#include <boost/thread.hpp>
#include <iostream>

namespace {
   boost::mutex ausgabeMutex;
}

class Ablage {
public:
   Ablage()
      : belegt(false), beendet(false) {
   }
   int get() {
      boost::unique_lock<boost::mutex> lock(monitorMutex);
      while(!belegt) {
         cond.wait(lock);
         if(beendet) {
            throw "get()-wait beendet";
         }
      }
      belegt = false;
```

```
      cond.notify_one();
      return inhalt;
   }
   void put(int wert) {
      boost::unique_lock<boost::mutex> lock(monitorMutex);
      while(belegt) {
         cond.wait(lock);
         if(beendet) {
            throw "put()–wait beendet";
         }
      }
      belegt = true;
      inhalt = wert;
      cond.notify_one();
   }
   void beenden() { // Alle wait()-Aufrufe zwangsläufig beenden
      beendet = true;
      cond.notify_all();
   }
private:
   bool belegt;
   bool beendet;
   int inhalt;
   boost::mutex monitorMutex;
   boost::condition_variable cond;
};
#endif
```

Auf den ersten Blick scheint es, dass ein mit `get()` wartender Consumer das Lock blockiert, sodass auch kein Producer Zugriff hat – schließlich beziehen sie sich beide auf dieselbe Mutex-Variable. Tatsächlich ist es nicht so, weil `wait()` eine wichtige Eigenschaft hat:

- Beim Aufruf von `wait()` wird implizit `monitorMutex.unlock()` aufgerufen, also die Sperre *gelöst*! Dann geht der Thread in den Zustand »wartend«.
- Nach einem `notify()` wird implizit `monitorMutex.lock()` aufgerufen, also das Lock angefordert. Erst wenn dies gelingt, also kein anderer Thread das Lock besitzt, kehrt `wait()` zurück.

Beide Vorgänge laufen atomar ab. Damit wird erreicht, dass ein wartender Thread das Objekt nicht gänzlich blockiert, und dass *nach* `wait()` kein anderer Thread im Objekt aktiv ist.

wait() unterbrechen

Wie gesagt, kann der Consumer nicht wissen, wann seine Aufgabe beendet ist, weswegen eine Exception zum Anhalten vorgesehen ist. Oder der Producer bleibt aufgrund eines Fehlers im Consumer hängen. Deswegen ist eine Möglichkeit, `wait()` unterbrechen zu können, sehr sinnvoll. In Abschnitt 13.4 wird dies mit dem Aufruf von `interrupt()` für einen Thread demonstriert. Ohne Interrupt ist dies genauso möglich, wenn es ein Flag `beendet` gibt, das von der Methode `beenden()` gesetzt wird. Anschließend muss sie die wartenden Threads freigeben, wie oben in der Klasse `Ablage` zu sehen.

Nach der Rückkehr aus `wait()` wird das Flag abgefragt, und es wird, falls gesetzt, eine Exception geworfen, die der Aufrufer der Funktion auffangen kann. Verlassen der Funktion mit einer Exception bewirkt den Aufruf der Destruktoren aller lokal angelegten Objekte, in diesem Fall nur der Variablen `lock`. Der Destruktor sorgt für die Freigabe der Mutex-Variablen, sodass andere wartende Threads befreit werden, ihrerseits `wait()` verlassen und sich auf dieselbe Art beenden.

Übung

13.1 Erweitern Sie das Producer-Consumer-Beispiel:
1. Die Ablage soll nicht nur einen, sondern mehrere Plätze haben. Diese sollen intern durch ein `int`-Array implementiert werden. Die Anzahl der Plätze wird dem Konstruktor mitgegeben.
2. Zuerst abgelegte Objekte sollen auch zuerst wieder entnommen werden (FIFO).
3. Es sollen mehrere Producer und mehrere Consumer gleichzeitig arbeiten können. Hinweis: Statt das Attribut `belegt` auszuwerten, ist es sinnvoll, den Zustand voll bzw. leer abzufragen.

13.6 Reader/Writer-Problem

Im vorhergehenden Abschnitt haben Sie gesehen, wie der gleichzeitige Zugriff auf eine Ressource gegenseitig verriegelt wird. Das ist manchmal nicht erwünscht. Eine Datei oder eine Datenbank (die Ressource) soll von vielen gleichzeitig gelesen werden können. Nur wenn eine Änderung notwendig wird, müssen zur Sicherung der Integrität der gelesenen Informationen alle Lesezugriffe gesperrt werden, bis die Änderung vollzogen ist. Der gegenseitige Ausschluss (englisch *mutual exclusion*) ist dafür zu einschränkend und kann die Performanz stark beeinträchtigen.

- Beliebig viele *Reader*, aber kein einziger *Writer* dürfen gleichzeitig aktiv sein. Ein geteilter Zugriff (englisch *shared access*) muss für Reader erlaubt sein.
- Wenn ein *Writer* aktiv ist, darf zur selben Zeit kein weiterer Writer und auch kein Reader aktiv sein. Ein Writer benötigt exklusiven Zugriff.

Solange die Integrität der Daten jedem lesenden Thread garantiert wird, also während des Lesens kein anderer Thread die Daten modifiziert, ist alles in Ordnung. Beliebig viele Reader oder ein einziger Writer können auf die Daten zugreifen, aber nie zur selben Zeit. Das wird durch ein Read-Write-Lock bewerkstelligt, in der Boost Library durch ein `shared_mutex` gesteuert. Zusammengefasst:

```
shared_mutex rwMutex;
shared_lock<shared_mutex> lock(rwMutex);  // Lock für geteilten Zugriff
unique_lock<shared_mutex> lock(rwMutex);  // Lock für exklusiven Zugriff
```

In der folgenden Klasse `Ressource` wird die Read-Write-Steuerung auf diese Weise realisiert. Das Attribut `inhalt` ist ein schlichter String, der gelesen bzw. geschrieben wird. Der Schwerpunkt des Beispiels liegt auf der Dokumentation der Zugriffe, damit erkenn-

bar wird, wie viele Reader bzw. Writer gerade aktiv sind. Um wirklich zu erreichen, dass mehrere Reader gleichzeitig aktiv sind, wird der lesende Zugriff zeitlich verzögert. Auch der Writer wird gebremst.

Listing 13.16: Ressource

```
// cppbuch/k13/rw/Ressource.h
#ifndef RESSOURCE_H
#define RESSOURCE_H
#include <string>
#include <boost/thread.hpp>
#include<boost/lexical_cast.hpp>
#include <iostream>
#include<Random.h>

namespace {
   boost::mutex ausgabeMutex;
   Random derZufall(500);
}

class Ressource {
public:
   Ressource()
      : inhalt("nichts"), nreader(0), nwriter(0) {
   }

   std::string read(const std::string& id) {
      // Lock für geteilten Zugriff:
      boost::shared_lock<boost::shared_mutex> lock(rwMutex);
      ++nreader;
      println(id + " liest " + inhalt
                             + rwprotokoll()); // siehe private Methode unten
      // lesen dauert ... (Simulation)
      boost::this_thread::sleep(
                          boost::posix_time::millisec(2000+ derZufall()));
      --nreader;
      return inhalt;
   }

   void write(const std::string& neu, const std::string& id) {
      // Lock für exklusiven Zugriff:
      boost::unique_lock<boost::shared_mutex> lock(rwMutex);
      ++nwriter;          // mögliche Werte 0 oder 1
      println(id + " schreibt " + neu + rwprotokoll());
      // schreiben dauert auch ... (Simulation)
      boost::this_thread::sleep(
                          boost::posix_time::millisec(1000+ derZufall()));
      --nwriter;
      inhalt = neu;
   }

   static void println(const std::string& was) {
      boost::lock_guard<boost::mutex> lock(ausgabeMutex);
```

```
      std::cout << was << std::endl;
   }
private:
   std::string rwprotokoll() {
      std::string msg(" Anzahl Aktiver: R=");
      // lexical_cast wandelt die Zahl in einen string um, siehe Seite 629
      msg += boost::lexical_cast<std::string>(nreader) + " W="
          + boost::lexical_cast<std::string>(nwriter);
      return msg;
   }
   std::string inhalt;
   boost::shared_mutex rwMutex;
   int nreader;
   int nwriter;
};

#endif
```

Im `main()`-Programm werden mehrere Reader und zwei Writer angelegt, die alle um die Ressource konkurrieren:

Listing 13.17: Reader-Writer-Beispiel

```
// cppbuch/k13/rw/main.cpp
#include<boost/thread.hpp>
#include<functional> // ref()
#include"Ressource.h"
#include"Writer.h"
#include"Reader.h"

using namespace std;

int main() {
   Ressource ressource;
   Writer w1(ressource, "w1");
   Writer w2(ressource, "w2");
   Reader r1(ressource, "r1");
   Reader r2(ressource, "r2");
   Reader r3(ressource, "r3");
   Reader r4(ressource, "r4");

   boost::thread_group threads;
   threads.create_thread(std::ref(w1));
   threads.create_thread(std::ref(r1));
   threads.create_thread(std::ref(r2));
   threads.create_thread(std::ref(r3));
   threads.create_thread(std::ref(r4));
   boost::this_thread::sleep(boost::posix_time::seconds(1));
   threads.create_thread(std::ref(w2));

   // Threads eine Zeit lang laufen lassen
   boost::this_thread::sleep(boost::posix_time::seconds(30));
   w1.beenden();
   w2.beenden();
```

```
   r1.beenden();
   r2.beenden();
   r3.beenden();
   r4.beenden();
   threads.join_all(); // warten, bis alles beendet ist
}
```

Jetzt fehlen nur noch die Reader und Writer. Weil sie einiges gemeinsam haben, werden die gemeinsamen Methoden und Attribute in eine abstrakte Oberklasse `ReaderWriter` verlagert:

Listing 13.18: Oberklasse ReaderWriter

```
// cppbuch/k13/rw/ReaderWriter.h
#ifndef READERWRITER_H
#define READERWRITER_H
#include"Ressource.h"

class ReaderWriter {
public:
   virtual void operator()() = 0; // vom Thread aufgerufene Methode
   void beenden() {
      ende = true;
   }
protected:
   ReaderWriter(Ressource& r, const std::string& i)
      : ende(false), ressource(r), id(i) {
   }
   virtual ~ReaderWriter() {}
   bool ende;
   Ressource& ressource;
   std::string id;
};
#endif
```

Weil die Klassen `Reader` und `Writer` von `ReaderWriter` erben, gestalten sich die zugehörigen Dateien recht kurz. Außer dem Konstruktor muss nur die Methode `operator()()` definiert werden.

Listing 13.19: Klasse Reader

```
// cppbuch/k13/rw/Reader.h
#ifndef READER_H
#define READER_H
#include"ReaderWriter.h"

class Reader : public ReaderWriter {
public:
   Reader(Ressource& r, const std::string& i)
      : ReaderWriter(r, std::string("Reader ") + i) {
   }
   void operator()() {
      Random zufall(1000);
      while(!ende) {
```

```
        std::string inhalt = ressource.read(id); // hier nicht weiter verwendet,
                                                 // es geht nur um den reinen Lesevorgang.
        // Den nächsten Lesevorgang zufällig verzögern:
        boost::this_thread::sleep(boost::posix_time::millisec(zufall()));
     }
     Ressource::println(id + " beendet sich.");
   }
};
#endif
```

Listing 13.20: Klasse Writer

```
// cppbuch/k13/rw/Writer.h
#ifndef WRITER_H
#define WRITER_H
#include"ReaderWriter.h"

class Writer : public ReaderWriter {
public:
   Writer(Ressource& r, const std::string& i)
   : ReaderWriter(r, std::string("Writer ") + i) {
   }
   void operator()() {
      int nr = 0;
      Random zufall(200);
      while(!ende) {
         std::string nachricht("Nachricht Nr.");
         // lexical_cast wandelt die Zahl in einen string um, siehe Seite 629
         nachricht += boost::lexical_cast<std::string>(++nr);
         ressource.write(nachricht, id);
         // Den nächsten Schreibvorgang zufällig verzögern:
         boost::this_thread::sleep(boost::posix_time::millisec(zufall()));
      }
      Ressource::println(id + " beendet sich.");
   }
};
#endif
```

Am besten lassen Sie das Programm mehrmals laufen. Durch das eingebaute Zufallselement, aber auch, weil dem Laufzeitsystem keine Vorgabe gegeben werden kann, wann welche Threads laufen, ist das Ergebnis variabel. Ein möglicher Auszug:

Reader r1 liest nichts Anzahl Aktiver: R=2 W=0
Writer w1 schreibt Nachricht Nr.1 Anzahl Aktiver: R=0 W=1
Reader r4 liest Nachricht Nr.1 Anzahl Aktiver: R=1 W=0
Reader r3 liest Nachricht Nr.1 Anzahl Aktiver: R=2 W=0
Reader r2 liest Nachricht Nr.1 Anzahl Aktiver: R=4 W=0
Reader r1 liest Nachricht Nr.1 Anzahl Aktiver: R=4 W=0
Writer w2 schreibt Nachricht Nr.1 Anzahl Aktiver: R=0 W=1

13.6.1 Wenn Threads verhungern

Da keine Annahmen getroffen werden können, welcher Thread wann läuft, können die folgenden Situationen entstehen:

- Es kommen ständig neue Reader hinzu, auch wenn andere den Lesevorgang beendet haben, sodass die Anzahl stets größer als 0 ist. In diesem Fall hat kein Writer eine Chance, die Daten zu aktualisieren – man sagt, der Writer-Thread verhungert.
- Ein oder mehrere Writer haben jeweils nacheinander exklusiven Zugriff, sodass kein Reader eine Chance zum Lesen der Daten bekommt. Die Reader »verhungern«.

Es gibt also kein Fair Play in dem obigen Algorithmus. Reader und Writer konkurrieren gnadenlos. Daraus ergibt sich die Frage: Wenn sowohl Reader als auch ein Writer gleichzeitigen Zugriff begehren, wer sollte bevorzugt werden?

- Falls Reader bevorzugt werden, können gleichzeitig viele lesende Threads laufen – die Parallelität und damit die Ausnutzung der Ressource steigt. Die Gefahr: Ein Writer könnte verhungern.
- Falls ein Writer bevorzugt wird, wird die Parallelität wegen des exklusiven Zugriffs reduziert. Reader könnten verhungern, obwohl die Wahrscheinlichkeit dafür in der Praxis klein ist. Meistens gibt es viele Reader und nur wenige oder einen Writer. Ein Vorteil der Bevorzugung eines Writers ist, dass die Daten schneller aktualisiert werden.
- Wenn die Aktualität der Daten hohe Priorität hat, kann man den Zutritt neuer Reader blockieren, sobald ein Writer Zugriff auf die Ressource verlangt, sodass der Writer sofort nach Ende des letzten Lesevorgangs die Daten aktualisieren kann.

Allerdings können den Threads keine Prioritäten zugeordnet werden, man muss also bei Bedarf eine andere Lösung wählen. Um den dritten Fall zu realisieren, kann man den Writern ermöglichen, sich anzumelden. Sobald mindestens ein Writer angemeldet ist, müssen neue Reader warten. Dazu braucht es neue Attribute in der Klasse `Ressource`:

```
int angemeldeteWriter;
boost::mutex writerAnmeldungsMutex;
```

Der Mutex garantiert, dass der gleichzeitige Zugriff von mehr als einem Writer auf die Variable `angemeldeteWriter` keine Inkonsistenz erzeugt. Die damit modifizierte Methode `write()` kann jetzt von vielen Writern betreten werden. Bis auf einen bleiben sie am `unique_lock` hängen, bis der erste den Schreibvorgang beendet hat:

```
// Auszug aus cppbuch/k13/rwp/Ressource.h
  void write(const std::string& neu, const std::string& id) {
  {
    boost::lock_guard<boost::mutex> anmeldungsLock(writerAnmeldungsMutex);
    ++angemeldeteWriter;
  }   // Lock wird freigegeben, sodass sich weitere Writer anmelden können.
  // Ab hier muss ggf. gewartet werden:
  boost::unique_lock<boost::shared_mutex> lock(rwMutex);
  ++nwriter;
  println(id + " schreibt " + neu + rwprotokoll());
  boost::this_thread::sleep(boost::posix_time::millisec(1000 + derZufall()));
  --nwriter;
  inhalt = neu;
  boost::lock_guard<boost::mutex> anmeldungsLock(writerAnmeldungsMutex);
```

```
   --angemeldeteWriter;       // abmelden
}
```

Auch das Herunterzählen muss verriegelt werden. Für `anmeldungsLock` ist im Gegensatz zum Eingang der Methode kein eigener Block erforderlich, weil die Methode zwei Zeilen später beendet und damit das Lock freigegeben wird. In der Methode `read()` wird die Information über die Anzahl der Anmeldungen ausgewertet:

```
std::string read(const std::string& id) {
   while(angemeldeteWriter > 0) {
      boost::this_thread::sleep(boost::posix_time::millisec(500)); // siehe Text
   }
   boost::shared_lock<boost::shared_mutex> lock(rwMutex);
   ++nreader;
   println(id + " liest " + inhalt + rwprotokoll());
   boost::this_thread::sleep(boost::posix_time::millisec(2000 + derZufall()));
   --nreader;
   return inhalt;
}
```

Die `while`-Schleife am Anfang sorgt dafür, dass kein Reader bei angemeldeten Writern versucht, das `shared_lock` in Anspruch zu nehmen. Ob die Bedingung noch gilt, wird periodisch überprüft. Auf Seite 433 wird empfohlen, Polling nicht einzusetzen, wenn es eine Alternative mit `wait` gibt. Diese Alternative ist in diesem Fall nicht oder nur schwierig zu realisieren. Nehmen Sie an, in der Schleife gäbe es ein `wait()`, etwa

```
while(angemeldeteWriter > 0) {
  condition.wait(lock);
}
```

Dann müsste, sobald das Ende der Methode `write()` erreicht wird, ein `notify_all()` erfolgen, um die Reader zu befreien. Das Problem: Es kann sein, dass dies *nach* Zeile 1, aber *vor* Zeile 2 geschieht. Ein `notify_all()` vor dem `wait()` ist jedoch wirkungslos! Da ab diesem Zeitpunk möglicherweise kein anderer Writer mehr aktiv wird, würden die Reader nie befreit. Die Alternative, die Schleife atomar auszuführen, indem sie mit einem Lock versehen wird, ist keine, weil erstens viele Reader in der Schleife warten sollen und zweitens mit demselben Lock das Herunterzählen von `angemeldeteWriter` samt `condition.notify_all()` versehen werden müsste. Damit wäre ein Deadlock vorprogrammiert.

13.6.2 Reader/Writer-Varianten

Eine ganze Datenbank wegen eines einzelnen Writer-Zugriffs für alle Reader zu sperren, ist bei stark durch lesende Anfragen belastete Datenbanken nicht akzeptabel. Deswegen ist es üblich, verschiedene Grade von Sperrungen zu ermöglichen, wobei zwischen Aufwand und Wirkung abgewogen werden muss. So kann bei einem Update nur eine Tabelle gesperrt werden oder sogar nur ein Record. Dasselbe Problem tritt nicht nur bei Datenbanken, sondern bei allen komplexen Datenstrukturen auf, wenn sie intensiv genutzt werden. Je feiner granular die Sperrung ist, desto höher ist der Verwaltungsaufwand. Die dadurch sinkende Performanz zahlt sich aus, wenn die Parallelität der Zugriffe nur wenig eingeschränkt wird. Ein allgemeingültiges Rezept gibt es nicht; die tatsächliche Nutzung im Einzelfall muss betrachtet werden.

13.7 Thread-Sicherheit

Threads bringen durch die Parallelität große Vorteile, genügend Prozessoren vorausgesetzt. Wenn es um gemeinsame Daten geht, ist der Synchronisationsaufwand teilweise erheblich, aber unerlässlich. Der Begriff *Thread-Sicherheit* einer Klasse bedeutet, dass alle Methoden korrekt ausgeführt werden, unabhängig von der Anzahl der ausführenden Threads. Das bedeutet auch, dass der Zustand eines Objekts dieser Klasse, gegeben durch die Werte seiner Attribute, stets konsistent (widerspruchsfrei) bleibt, und zwar ohne dass Benutzer der Klasse selbst dafür sorgen müssen. Die Klasse stellt intern alle notwendigen Synchronisationsmechanismen zur Verfügung. Bei dem Entwurf so einer Klasse ist zu berücksichtigen:

- Es dürfen keinerlei Annahmen getroffen werden, in welcher Reihenfolge Threads vom Betriebssystem zur Ausführung gebracht werden. Keine der möglichen Reihenfolgen darf das Objekt in einem ungültigen Zustand hinterlassen. Anders gesagt: Es darf keine Race Conditions geben, also Bedingungen, unter denen das Programmergebnis vom Timing oder der Abfolge von Threads abhängt.
- Zusammengehörende Operationen wie das Lesen und Verändern einer gemeinsam genutzten Variablen müssen *atomar* ausgeführt werden (siehe Beispiel auf Seite 426).
- Thread-sichere Software muss extrem sorgfältig entworfen werden, weil vollständige Tests auf Thread-Sicherheit sehr schwierig bis unmöglich sind. In der Regel können nicht alle möglichen Abfolgen von Threads überprüft werden. So kann es sein, dass ein Programm zwar alle Tests besteht, jedoch in einigen Jahren ein Fehler auftritt, weil zufällig zwei Threads unsynchronisiert dieselbe Variable ändern.
- Ebenso sorgfältig muss die Reihenfolge der Akquirierung und Freigabe von Ressourcen geplant werden, damit kein Deadlock (Verklemmung) entsteht. Ein Deadlock bewirkt, dass das Programm stehen bleibt. Eine typische Ursache: Thread A akquiriert Ressource X, Thread B akquiriert Ressource Y. Nun möchte A ebenfalls Y akquirieren, und Thread B die Ressource X. Weil die Ressourcen vorher nicht freigegeben wurden, warten beide bis zum Jüngsten Tag aufeinander oder bis jemand das Programm mit Strg+C abbricht.

Im Zusammenhang mit der Thread-Sicherheit wird gelegentlich der Begriff *reentrant* (dt. etwa wiedereintrittsfähig) genannt. Darunter ist zu verstehen, dass eine Funktion gleichzeitig ohne Probleme von verschiedenen Threads aufgerufen werden kann, wenn keine gemeinsamen Daten benutzt werden. Die Voraussetzungen sind bei vielen modernen Programmiersprachen gegeben: Der ausführbare Programmcode liegt in einem schreibgeschützen Bereich (kann also nicht geändert werden), und jedem Aufruf wird ein eigener lokaler Speicherbereich für die Daten zugewiesen.

In diesem Kapitel werden die Grundzüge der Arbeit mit C++-Threads dargestellt. Mehr über Boost-Threads erfahren Sie in [thread]. Wenn Sie mehr über die Anwendung von Software-Mustern in der Programmierung paralleler Anwendungen wissen möchten, kann ich Ihnen das Buch [SSRB] empfehlen. Grundlagen über die Parallelität von Prozessen, Synchronisation, Deadlock-Entdeckung und -Vermeidung finden Sie in Büchern, in deren Titel »Parallele Programmierung«, »concurrent programming« oder »operating system principles« vorkommen.