Beispiel `07_berkeley`

Dr. Günter Kolousek

31. August 2020

1 Allgemeines

- Im ersten Beispiel gibt es genaue Anweisungen zum Aufbau und der Durchführung eines Beispiels. Bei Bedarf nochmals durchlesen!
- Trotzdem hier noch zwei Erinnerungen:
  - **Regelmäßig** Commits erzeugen!
  - **Backup** nicht vergessen!
- In diesem Sinne ist jetzt ein neues Verzeichnis `07_berkeley` anzulegen.

2 Aufgabenstellung

Aufgabe ist es, ein Programm zur Simulation des Berkeley Algorithmus zu schreiben. Dieses Beispiel soll hauptsächlich das Verständnis für den Berkeley Algorithmus verbessern, aber auch der Umgang mit Streams und das Entwerfen von Klassen wird geübt. Entwurfsrichtlinien:

- Die Fehlerbehandlung steht derzeit nicht im Vordergrund. D.h. derzeit kann man ruhig noch Konstrukte der folgenden Art als exception handler verwenden:

```
// header: <exception>
// all exceptions inherit from std::exception
catch (const std::exception& e) {
    cerr << e.what() << endl;
}
```

- Das Beispiel ist auf 2 Clients beschränkt.
- Wie immer: Achte darauf, dass Instanzvariablen nur in Ausnahmefällen `public` deklariert werden!!! D.h. wenn es sinnvoll ist.

Los geht's!

3 Anleitung

1. Schreibe eine Klasse `Clock` (in einem eigenen Modul `clock`), die eine Uhr simuliert.

   Es soll vorerst nur ein Konstruktor vorhanden sein:

   ```
   // Name der Uhr; aktueller Tag und aktuelle Zeit der Systemuhr
   Clock(std::string name);
   ```

   Danach wird in einem eigenen Thread die Zeit weitergestellt. Anfangs lassen wir die Uhr synchron zur Rechneruhr laufen (d.h. jede Sekunde wird die eigene Uhr um 1 Sekunde vorgestellt).

   Jede "Sekunde" (d.h. die Sekunde deiner Uhr, nicht die der Rechneruhr, siehe später) soll die Uhrzeit gemeinsam mit dem Uhrennamen ausgegeben werden.

   Die Ausgabe findet *in diesem Beispiel* direkt nach dem Weiterstellen der Uhr statt. Der Sinn ist, dass wir die Uhr leicht testen können ohne in Gefahr einer Race Condition zu gelangen.

   Teste diese Uhr in einem Hauptprogramm mit einem Thread, was im Code so aussehen sollte:

   ```
   thread clock{Clock("testclock")};
   clock.join();
   ```

   Die Ausgabe sollte dann folgendermaßen aussehen:

   ```
   testclock: 2015-12-03 12:18:38
   testclock: 2015-12-03 12:18:39
   testclock: 2015-12-03 12:18:40
   ...
   ```

   - Verwende dazu die Klasse `std::chrono::system_clock`. Diese hat eine statische Methode `now()`, die die aktuelle Zeit als Instanz der Klasse `std::chrono::time_point<std::chrono::system_clock>` zurückliefert.
   - Tipp: Verwende so oft wie möglich das Keyword `auto`, allerdings geht dies nicht bei der Deklaration einer Instanzvariable so wie wir das gerne hätten. Also helfe ich einmal aus:

     ```
     chrono::time_point<chrono::system_clock> curr_time;
     ```

     Damit wird eine Variable definiert, die einen Zeitpunkt repräsentiert, der auf der Systemuhr basiert und die Defaultgenauikeit aufweist. Standardmäßig hat die Systemuhr meist eine Genauigkeit im Nanosekundenbereich.

     Wollte man eine andere Genauigkeit haben, wie z.B. im Sekundenbereich dann müsste dies folgendermaßen aussehen:

     ```
     chrono::time_point<chrono::system_clock, chrono::seconds> curr_time;
     ```

Genauer: Es wird ein Zeitpunkt angegeben, der als Sekunden (als `duration`) vom Beginn der Epoche (der `system_clock`) angegeben wird.

Wir sind aber mit der Defaultgenauigkeit zufrieden. Allerdings wäre diese noch mit der aktuellen Zeit zu initialisieren, aber das kannst du selber.

- Jetzt geht es darum, die Zeit auszugeben.

  Ich stelle hierfür eine Headerdatei `timeutils.h` zur Verfügung, In dieser ist ein Operator `<<` definiert, der es erlaubt eine `time_point` Instanz in der gewünschten Form auszugeben.

  Dazu ist lediglich die Headerdatei zu inkludieren. Es schadet aber nicht einen Blick in die Datei zu riskieren.

  Damit wird die Ausgabe eines Zeitpunktes prinzipiell so erscheinen wie oben gezeigt.

- Im Operator `()`, der wiederum als *eigener* Thread (gestartet im Hauptprogramm) laufen soll, wird von der ermittelten Zeit im 1s Takt die ermittelte Zeit um jeweils einer Sekunde weitergestellt (also in einer Endlosschleife). D.h. in einer Schleife jeweils eine Sekunde warten und dann zur aktuellen Zeit `1s` (eine Instanz von `duration`) addieren und das Ergebnis wieder als aktuelle Zeit speichern und ausgeben (es gibt auch einen überladenen Operator `+=` in der Klasse `time_point`).

  Jetzt bekommst du so eine Ausgabe wie oben gezeigt.

2. Erweitere deine Klasse `Clock` um einen zweiten Konstruktor, der für den aktuellen Tag die Zeit in Stunden, Minuten und Sekunden setzt:

   ```
   Clock(string name_, int hours_, int minutes_, int seconds_);
   ```

   Da man nicht direkt die Stunden, Minuten und Sekunden bei einem Zeitpunkt setzen kann, habe ich in der Headerdatei `timeutils.h` eine Funktion `set_time` zur Verfügung gestellt, die einen `time_point` sowie die Stunden, Minuten und Sekunden als Parameter erhält und einen neuen `time_point` zurückliefert.

3. Erweitere jetzt die Klasse `Clock` um die beiden folgenden Methoden, die die Uhrzeit setzen (analog zum Konstruktor) bzw. abfragen:

   ```
   void Clock::set_time(int hours, int minutes, int seconds);
   tuple<int, int, int> Clock::get_time();
   ```

   Auch hier kannst du auf die Funktionen aus der bereitgestellten Headerdatei `timeutils.h` zurückgreifen.

   Beachte, dass es jetzt eine Funktion `set_time` und eine Methode `set_time` der Klasse `Clock` gibt. Um eine Funktion mit dem gleichen Namen aufzurufen benötigst du den scope resolution operator!

   Was darf man auf keinem Fall vergessen?

4. Schreibe eine Klasse `TimeSlave`, die als Thread einen Rechner simuliert. Beim Instanzieren wird ein Rechnername und eine Anfangszeit mitgegeben. Diese Instanz soll sich eine eigene Uhr anlegen (mit der übergebenen Zeit und dem Rechnernamen als Uhrenname). Der Operator `operator()` wird später den Berkeley-Algorithmus implementieren.

   Diese Klasse kann durchaus in der Datei `main.cpp` definiert werden. Es geht ja hier nur um das Prinzip und eigene Module haben wir ja schon geübt und können dies schon.

5. Erweitere jetzt das Hauptprogramm so, dass zwei Instanzen von `TimeSlave` angelegt werden und diese mit einem Namen (”slave1” und ”slave2” wären nicht ganz verkehrt oder ähnliches) und einer Anfangszeit angelegt und danach in einem Thread gestartet werden.

   Vergiss nicht, dass auch die Ausgabe synchronisiert werden muss.

6. Schreibe nun eine Klasse `TimeMaster` analog zu der Klasse `TimeSlave`, also gleiche Funktionalität. Auch diese Klasse kann direkt in `main.cpp` definiert werden.

   Erweitere auch gleich das Hauptprogramm um eine Instanz dieser Klasse analog zu den `TimeSlave`-Instanzen.

   Damit wird beim Starten des Programmes schon einiges ausgegeben.

7. Nimm die Datei `pipe_template.h` und kopiere diese auf `pipe.h`. Dann erweitere die Datei so, dass eine Pipe entsteht, die im Kontext mehrerer Threads verwendet werden kann. Diese Pipe soll in gewissen Grenzen so funktionieren wie ein Stream in `C++` funktioniert, wobei es sich allerdings um einen *parametrisierten* Typen handelt.

   - Der Operator `<<` wird verwendet, um einen Wert in die Pipe zu stellen.
   - Der Operator `>>` wird verwendet, um einen Wert aus der Pipe zu lesen.
   - Die Methode `close` schließt die Pipe, sodass keine weiteren Werte mehr in die Pipe gestellt werden können und auch keine weiteren Werte aus der Pipe gelesen werden können.

     Werden Werte in die Pipe gestellt bzw. aus dieser gelesen, wenn die Pipe schon geschlossen wurde, dann ist das Verhalten nicht definiert.
   - Der Operator `bool` kann verwendet werden, sodass diese Pipe wie ein Stream verwendet werden kann, also so etwas wie:

     ```
     while (pipe >> value) {
     ```

     Dazu muss dieser Operator `bool` anzeigen, ob die Pipe noch offen ist.

8. Schreibe eine Klasse `Channel`, die über zwei Pipes verfügt, die jeweils `long` Werte übertragen können.

Jede Pipe soll für eine Richtung der Kommunikation zwischen Slave und Master verwendet werden. D.h. je ein Channel für die bidirektionale Kommunikation zwischen Master und und einem Slave.

Die Klasse `Channel` soll über die beiden zwei Methoden `Pipe<long>& get_pipe1()` und `Pipe<long>& get_pipe2()` verfügen.

Auch diese Klasse kann der Einfachheit halber in `main.cpp` definiert werden.

9. Erweitere die Klasse `TimeSlave` um eine Instanzvariable für einen Channel und eine Methode zu Zugriff darauf: `Channel* get_channel()`.

10. Erweitere die Klasse `TimeMaster` um zwei Instanzvariablen zum abspeichern der Zeiger auf je einen Channel (pro `TimeSlave` Instanz) und eine Methode `void set_channel1(Channel*)` zum Setzen dieses Pointers auf den Channel zum Slave 1. Analog dazu soll das Setzen des Channels zum Slave 2 funktionieren.

11. Jetzt ist es an der Zeit diese Teile zu testen, indem du den Master mit den Slaves verbindest.

    Der Master schickt von sich aus je 3 Werte an die Slaves schickst und schließt nach einer kurzen Wartezeit von 500ms (warum?) die jeweiligen Pipes.

    Im Slave werden die Werte in einer Schleife ausgelesen (solange die Pipe noch nicht geschlossen ist) und ausgegeben.

12. Erweitere jetzt `TimeMaster` als auch `TimeSlave`, sodass diese den Berkeley Algorithmus implementieren und sinnvolle Ausgaben durchführen (empfangene, berechnete und gesendete Werte).

    Setze dazu die Anfangszeiten von Master, Client 1 und Client 2 verschieden.

    Übertrage die Uhrzeit in Sekunden (als Anzahl der Sekunden vom 1.1.1970) (auch wenn es verschiedene Epochen gibt, beginnt die klassische Epoche am 1.1.1970 um 00:00:00 UTC). Das kann folgendermaßen erreicht werden:

    - Die Klasse `system_clock` hat eine statische Methode (engl. static member function) `time_t to_time_t(const time_point&)`, die aus einem `time_point` einen Wert vom Typ `time_t` ermittelt, der die Anzahl der Sekunden seit Beginn der Epoche angibt.
    - Die Klasse `system_clock` hat außerdem eine statische Methode `chrono::system_clock::time_p` `from_time_t(std::time_t)`, die einen `time_point` aus dem übergebenen `time_t` erzeugt.

    Erweitere dazu deine Klasse `Clock` um die Methoden `long to_time()` und `void from_time(long)`, die die oben genannten statischen Methoden verwenden. Der Typ von `time_t` ist im Standard der Programmiersprache C nicht definiert. Es ist lediglich festgehalten, dass es sich um einen arithmetischen Typen handelt. Wir

können aber davon ausgehen, dass dieser in der Regel einem `int` entspricht. Diese Lösung funktioniert im Jahr 2038 nicht mehr!

Der Berkeley Algorithmus soll alle 10 Sekunden durchgeführt werden. Das können ruhig "echte" 10s sein (also die Sekunden deines Rechners).

13. Erweitere jetzt noch die Klasse `Pipe` um eine Methode `void set_latency(long)`, die eine Instanzvariable `latency` auf den Wert setzt. Weiters soll das Schreiben eines Wertes in die Pipe um diesen Wert verzögert werden, wenn die Latenz ungleich 0 ist.

    Dazu solltest du einen `async` Aufruf wagen, sodass in einem eigenen Thread der Wert um die Latenz verzögert in die Backend-Datenstruktur geschrieben wird.

    Weiters ist die Klasse `Channel` um eine `set_latency` Methode zu erweitern, die den übergebenen Wert an beide Pipes weitergibt.

    Verwende für die beiden Channels jeweils verschiedene Latenzen.

14. Modifiziere die Klasse `Clock` so, dass diese unterschiedlich "schnell" gehen kann. D.h. anstatt immer 1000ms zu warten (je Sekundenschritt) kann man z.B. 980ms warten (Uhr geht schneller) oder z.B. 1200ms (Uhr geht langsamer). Siehe in weiterer Folge Punkt 16 (Abweichungen vom Sekundentakt → Ungenauigkeit)!

15. Modifiziere den Master als auch die Slaves so, dass die Uhr **nicht** mehr zurückgestellt wird. Schreibe dazu eine Methode `void set_time_monoton(bool)` in der Clock Klasse. Damit soll die Uhr nicht mehr nur um die Abweichung falsch gehen, sondern das Weiterschalten nicht in einer Sekunde sondern z.B. in 2 Sekunden erfolgen (plus die Abweichung).

16. Erweitere das Programm, sodass es folgende Benutzerschnittstelle implementiert:

```
Simulate the berkeley-algo
Usage: berkeley [OPTIONS]

Options:
  -h,--help                   Print this help message and exit
  --monotone                  set monotone mode
  --latency1 UINT             latency to channel 1 (both directions)
  --latency2 UINT             latency to channel 1 (both directions)
  --deviation1 INT            deviation of clock of slave 1
  --deviation2 INT            deviation of clock of slave 1
  --deviationm INT            deviation of clock of master
```

4 Übungszweck

- `<chrono>` kennenlernen

- Simulation einer Uhr üben und erstes Verständnis für Uhren
- Implementieren von thread-safe Datenstrukturen üben
- Streams üben
- Berkeley Algo verstehen (auch bei Latenzen, unterschiedlichen Uhren,…)