Beispiel `05_sync3`

Dr. Günter Kolousek

7. September 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 `05_sync3` anzulegen.

2 Aufgabenstellung

Aufgabe ist es, ein Programm zur Simulation der "Dining philosophers" zu schreiben. Das Problem lässt sich folgendermaßen formulieren: An einem runden Tisch sitzen 5 Philosophen und tun das, was Philosophen zu tun pflegen: denken und essen (in Wirklichkeit nicht notwendigerweise in dieser Reihenfolge). In der Mitte steht ein großer Topf Spaghetti und zwischen je zwei Philosophen befindet jeweils eine Gabel. Zum Essen benötigt jeder Philosoph zwei Gabeln.
Los geht's!

3 Anleitung

1. Schreibe eine triviale Implementierung für dieses Problem. D.h. jeder Philosoph nimmt sich immer zuerst die Gabel zu seiner linken Hand und *unmittelbar* danach die rechte Gabel. Gedacht wird eine Sekunde lang und gegessen wird 2 Sekunden lang.

   Schreibe dazu zuerst eine Klasse `Philosopher`, dessen Konstruktor eine linke und eine rechte Gabel bekommt (neben der Nummer des Philosophen natürlich). Solch eine Instanz soll als Thread gestartet werden können. Implementiere dafür ein

Modul, also `philosopher.h` und `philosoper.cpp`, wobei in der Headerdatei nur der Konstruktor implementiert werden soll. In diesem Beispiel soll mit den Mutex-Instanzen direkt gearbeitet werden (also händisch `lock()` und `unlock()` aufrufen).

Dazu ist ein Programm `dining` zu entwickeln, das sich um die Erzeugung der Gabeln (realisiert mittels Mutex) kümmert und die Philosophen mit diesen Gabeln anlegt (und damit startet).

Abgebrochen wird das Programm mit `Ctrl-C`.

Die Ausgabe sollte folgendermaßen aussehen:

```
Philosopher 2 is thinking...
Philosopher 3 is thinking...
Philosopher 5 is thinking...
Philosopher 1 is thinking...
Philosopher 4 is thinking...
Philosopher 2 attempts to get left fork
Philosopher 3 attempts to get left fork
Philosopher 3 got left fork. Now he wants the right one...
Philosopher 1 attempts to get left fork
Philosopher 1 got left fork. Now he wants the right one...
Philosopher 2 got left fork. Now he wants the right one...
Philosopher 3 got right fork. Now he is eating...
Philosopher 4 attempts to get left fork
Philosopher 5 attempts to get left fork
Philosopher 5 got left fork. Now he wants the right one...
Philosopher 3 finished eating
Philosopher 3 released left fork
Philosopher 3 released right fork
Philosopher 3 is thinking...
```

Wenn die Ausgabe von der Struktur nicht so aussieht, sondern solche Artefakte enthält wie

```
Philosopher 3 is thinking...
Philosopher 4Philosopher 5 got left fork. Now he wants the right one... finished eat


Philosopher 4 got right fork. Now he is eating...
```

dann stellt sich die Frage warum? Alles klar?

Nein, dann frage deinen linken Sitznachbar oder sonst irgendwem… und bessere es aus!

Vergiss bitte nicht, das zu committen!

2. So, wie hast du jetzt das Problem gelöst, dass die Ausgaben auch synchronisiert werden? Vielleicht einen globalen Mutex genommen und jede Ausgabe mit einem `lock_guard` der folgenden Art geschützt:

```
{ lock_guard<mutex> lg{out_mtx};
  cout << "aus" << "gabe" << endl; }
```

Ja, prinzipiell richtig, aber etwas mühsam, nicht wahr? Besser wäre es da schon eine Funktion `println` zu schreiben, die folgendermaßen aufgerufen werden kann:

```
println({"aus", "gabe"});
```

Als Signatur bietet sich da folgende Form an:

```
void println(const vector<string>&);
```

Implementiere diese Funktion, die natürlich auf einen globalen Mutex (warum nicht wieder `out_mtx` nennen?) zugreift. Und dieser globale Mutex ist nun überhaupt nicht schlimm (für mich zumindest), da er ja dafür da ist, überall verwendet zu werden und es auch nur ein `cout` gibt.

Commiten nicht vergessen.

3. Warum muss eigentlich ein `vector<string>` erzeugt werden? Wäre es nicht sinnvoller eine `initializer_list<string>` in der Parameterliste zu verwenden? Doch, ist besser. Also, dann verbessere dein Programm in diesem Sinne. Und vergiss nicht auf das obligorische Commit!

   Gibt es einen Nachteil im Vergleich zur vorhergehenden Lösung? Jein. Na ja, einen `vector` kann man jetzt nicht mehr beim Aufrufen der Funktion verwenden, aber wer will so etwas schon.

4. Ok, so richtig elegant und performant ist das allerdings nicht, richtig? Warum?

   Denk, denk,…

   - Dass das nicht elegant ist, liegt auf der Hand, da es nicht sehr intuitiv ist, die geschwungenen Klammern beim Aufruf verwenden zu müssen.
   - Und performant ist es sicher auch nicht, da zur Laufzeit über die `initializer_list` iteriert werden muss.

   Was können wir da machen?

   Na ja, wir haben im Unterricht auch noch gehört, dass es da noch eine Möglichkeit von der Programmiersprache C gibt, eine variable Anzahl von Parametern zu verwenden. Allerdings haben wir auch besprochen, dass dies nicht typsicher ist (und auch nicht performanter).

   Ja, und da gibt es natürlich auch seitens `C++` eine Lösung, die allerdings auf Templates basiert (und die haben wir nicht gelernt). Deshalb hier von mir eine Lösung, die du gerne verwenden darfst / kannst / **sollst**:

```
std::recursive_mutex out_mtx;

void println() {
    std::lock_guard<std::recursive_mutex> lg{out_mtx};
    std::cout << std::endl;
}

template<typename T, typename... Rest>
void println(const T& word, const Rest&... rest) {
    std::lock_guard<std::recursive_mutex> lg{out_mtx};
    std::cout << word << ' ';
    println(rest...);
}
```

Den gesamten Codeblock kannst du der Einfachheit halber in `philosopher.cpp` kopieren und dann etwa so verwenden:

```
println("aus", "gabe");
```

Beim Übersetzen werden die Templates instanziert und der Compiler hat die Möglichkeit, die rekursiven Aufrufe wegzuoptimieren. Einen Nachteil gibt es trotzdem… Ja, jetzt müssen wir auf einmal einen rekursiven Mutex verwenden.

Diesen Code direkt in `philosopher.cpp` zu kopieren ist auch nicht besonders sinnvoll, besser wäre es diese Hilfsfunktionen samt der Variable in ein eigenes Modul auszulagern, wie z.B. `utils.h` und `utils.cpp`. Probiere, ob du es hinbekommst. Wenn ja, zeige es mir, wenn du der Erste/die Erste bist. Wenn nicht, dann macht es auch nichts.

5. Welches Problem kann auftreten? Füge zwischen dem Aufnehmen der linken Gabel und dem Aufnehmen der rechten Gabel eine genügend große Zeitspanne von 5s ein!

   Deine Ausgabe sollte jetzt so aussehen:

```
Philosopher 2 is thinking...
Philosopher 3 is thinking...
Philosopher 5 is thinking...
Philosopher 1 is thinking...
Philosopher 4 is thinking...
Philosopher 2 attempts to get left fork
Philosopher 2 got left fork. Now he wants the right one...
Philosopher 3 attempts to get left fork
Philosopher 3 got left fork. Now he wants the right one...
Philosopher 5 attempts to get left fork
Philosopher 5 got left fork. Now he wants the right one...
Philosopher 1 attempts to get left fork
Philosopher 1 got left fork. Now he wants the right one...
```

```
Philosopher 4 attempts to get left fork
Philosopher 4 got left fork. Now he wants the right one...
```

Also wie nennt man dieses Problem?

6. Als Vorarbeit auf den nächsten Punkt, ist jetzt in einem Modul eine Klasse `Semaphore` zu entwickeln, die einen klassischen zählenenden Semaphor implementiert.

   Zwei Konstruktoren (einmal mit Anfangswert und einmal ohne) und die Methoden `acquire`, `release` und `available_permits`! `available_permits` liefert den aktuellen Zählerstand zurück und die beiden anderen Methoden funktionieren wie man es von einem gutem Semaphor erwarten würde.

   In gut bewährter Weise sind nur die Konstruktoren in der Headerdatei zu implementieren. Was ist eigentlich der Unterschied zwischen dem Implementieren in einer Headerdatei und dem Implementieren in einer `.cpp` Datei?

   Verwende Locks und Bedingungsvariable.

7. Erweitere das Beispiel, sodass das Problem nicht auftritt. Es gibt dazu mehrere Möglichkeiten, wie z.B.:

   - Gleichzeitiges Aufnehmen der beiden Gabeln
   - Philosophen nehmen abwechselnd zuerst die linke und dann die rechte Gabel bzw. umgekehrt. Das kann z.B. leicht dadurch erreicht werden, dass die Philosophen durchnummeriert werden und alle ungeraden Philosophen die linke Gabel zuerst aufnehmen und alle geraden Philosophen die rechte Gabel zuerst.
   - Alle Gabeln werden durchnummeriert und jeder Philosoph nimmt sich immer zuerst die Gabel mit der niedrigeren Nummer. D.h. ein Philosoph nimmt sich die rechte Gabel zuerst, alle anderen die linke Gabel.
   - Es gibt einen Butler, der ihnen Gabeln zuteilt.
   - Philosophen stehlen sich eine Gabel!
   - Es gibt entweder 6 Gabeln oder es dürfen nicht mehr als 4 'linke' Gabeln gleichzeitig genommen werden.

   Verwende die letzte Möglichkeit. D.h. es dürfen nicht mehr als 4 'linke' Gabeln aufgenommen werden. Verwende dazu einen Semaphor.

   Übergebe einen *Pointer* auf den Semaphor, so besteht die Möglichkeit, dass man entweder einen `nullptr` oder einen Pointer auf eine konkrete Instanz einsetzen kann. Überprüfe im Thread, ob der Pointer ungleich `nullptr` ist und nur dann wird dieser verwendet. Damit haben wir im nächsten Punkt die Möglichkeit das Programm so zu erweitern, dass es sowohl mit als auch ohne Deadlock-Prävention läuft.

Die Ausgabe sollte jetzt folgendermaßen aussehen:

```
Philosopher 1 is thinking...
Philosopher 2 is thinking...
Philosopher 3 is thinking...
Philosopher 4 is thinking...
Philosopher 5 is thinking...
Philosopher 1 attempts to get left fork
Philosopher 1 got left fork. Now he wants the right one...
Philosopher 2 attempts to get left fork
Philosopher 4 attempts to get left fork
Philosopher 4 got left fork. Now he wants the right one...
Philosopher 2 got left fork. Now he wants the right one...
Philosopher 3 attempts to get left fork
Philosopher 3 got left fork. Now he wants the right one...
Philosopher 5 attempts to get left fork
currently 1 left forks available
Philosopher 4 got right fork. Now he is eating...
Philosopher 4 finished eating
Philosopher 4 released left fork
currently 2 left forks available
```

Beachte weiters, dass das Aufnehmen der linken Gabel und das Herunterzählen mittels des Semaphors eigentlich eine atomare Aktion sein müsste, wenn man will, dass der Zähler den richtigen Wert anzeigt. Für die Deadlock-Vermeidung ist das kein Problem. Aber wie könnte man dies lösen?

8. Ändere das Programm `dining` so ab, dass eine Kommandozeilenoption `--nodeadlock` mitgegeben werden kann, der angibt, dass kein Deadlock entstehen darf. Wird dieser nicht angegeben, dann wird keine Deadlockverhinderung verwendet.

   Hier werden wir wieder auf `CLI11` zurückgreifen! Allerdings ist es nicht sinnvoll jedes Mal die Datei `CLI11.hpp` in das Verzeichnis `include` zu speichern. Das ist einfach nicht sinnvoll. Meist ist es sinnvoller die verwendete Bibliothek irgendwo im System zu installieren und diese dann zu verwenden. Meson bietet hier Unterstützung in Form von Meson-Optionen: `meson_options.txt`:

   a) Kopiere dir die Datei `CLI11.hpp` irgendwo in dein Homeverzeichnis, aber eben nicht in ein Projektverzeichnis (d.h. nicht in dein Repository!).

   b) Ändere dann in der Datei `meson_options.txt` in der entsprechenden Zeile den Wert von `value` entsprechend ab indem du den absoluten Pfad von dem Verzeichnis einträgst, in das du die Datei `CLI11.hpp` abgelegt hast.

   c) Ändere jetzt noch die Datei `meson.build` entsprechend ab, sodass die Option `cli11_include_dir` aktiv wird.

   d) Lösche jetzt den Inhalt des Verzeichnisses `build` und übersetze dein Projekt nochmals.

Damit sollte einer korrekten Verwendung von `CLI11` nichts mehr im Wege stehen, ohne dass `CLI11.hpp` im Projekt abgelegt werden muss.

Hier wieder ein Beispiel, das die Funktionalität der TUI demonstrieren soll:

```
$ dining --help
Dining philosophers simulation
Usage: dining [OPTIONS]

Options:
  -h,--help                   Print this help message and exit
  -n,--nodeadlock             Prevent a deadlock at all
  -l,--livelock               Simulate a livelock
```

Die Option `--livelock` werden wir im nächsten Punkt noch benötigen und kann derzeit ignoriert werden.

9. Jetzt wollen wir unser Programm noch um das "Feature" eines Livelocks erweitern. Dazu bekommt das Programm eine weitere Kommandozeilenoption `--livelock`.

   Das eigentliche Feature wird so eingebaut, dass wir eine Regel einführen, die besagt, dass auf den rechten Lock nur 3s gewartet wird. Wird der Lock nicht innerhalb des Timeouts gehalten, dann wird nochmals 100ms gewartet (warum?) und die linke Gabel wieder zurückgelegt. Danach wird nochmals 3s gewartet und wieder mit der linken Gabel begonnen (also in einer Schleife).

   Die Ausgabe wird dann folgendermaßen aussehen:

```
Philosopher 4 is thinking...
Philosopher 2 is thinking...
Philosopher 1 is thinking...
Philosopher 5 is thinking...
Philosopher 3 is thinking...
Philosopher 4 attempts to get left fork
Philosopher 4 got left fork. Now he wants the right one...
Philosopher 3 attempts to get left fork
Philosopher 3 got left fork. Now he wants the right one...
Philosopher 5 attempts to get left fork
Philosopher 5 got left fork. Now he wants the right one...
Philosopher 2 attempts to get left fork
Philosopher 2 got left fork. Now he wants the right one...
Philosopher 1 attempts to get left fork
Philosopher 1 got left fork. Now he wants the right one...
Philosopher 4 released left fork due to timeout getting the right one!
Philosopher 3 released left fork due to timeout getting the right one!
Philosopher 5 released left fork due to timeout getting the right one!
Philosopher 2 released left fork due to timeout getting the right one!
Philosopher 1 released left fork due to timeout getting the right one!
```

```
Philosopher 4 attempts to get left fork
Philosopher 4 got left fork. Now he wants the right one...
Philosopher 3 attempts to get left fork
Philosopher 3 got left fork. Now he wants the right one...
Philosopher 2 attempts to get left fork
Philosopher 2 got left fork. Now he wants the right one...
Philosopher 5 attempts to get left fork
Philosopher 5 got left fork. Now he wants the right one...
Philosopher 1 attempts to get left fork
Philosopher 1 got left fork. Now he wants the right one...
```

Was passiert, wenn der Benutzer sowohl die Option `--nodeadlock` als auch `--livelock` eingibt?

4 Übungszweck dieses Beispiels

- Wiederholung der Locks
- Locks zur Repräsentation von Ressourcen verwenden.
- Synchronisation der Ausgabe erkennen und nicht vergessen.
- Funktion mit einer variablen Anzahl an Parametern auf der Basis eines Vektors implementieren.
- Wiederholung der impliziten Kovertierung, in diesem Fall von `initializer_list<string>` nach `vector<string>`, da `vector` eben einen entsprechenden Konstruktor hat (der nicht `explicit` ist).
- Unterschied zwischen `vector` und `initializer_list` betrachten.
- Variable Anzahl an Argumenten in `C++` verwenden.
- Globale Variable mittels `extern` kennzeichnen.
- Problematik Deadlock erkennen.
- Implementieren eines Semaphor auf Basis von Locks und Bedingungsvariablen.
- Deadlock verhindern, Arbeiten mit Semaphoren
- Wiederholung Zugriff auf Kommandozeilenargumente
- Meson-Optionen und `meson_options.txt` (speziell zur Verwendung von header-only Bibliotheken) kennenlernen
- Livelock kennenlernen und verstehen
- `timed_mutex` einsetzen