Verteilte Systeme

...für C++ Programmierer

Synchronisationsmechanismen

by

Dr. Günter Kolousek

Klassische Probleme

- Producer/Consumer
  - siehe Synchronisation
- Reader/Writer
  - mehrere dürfen gleichzeitig lesen
  - wenn einer schreibt, darf keiner lesen oder schreiben
  - siehe Read/Write Lock
- Dining philosophers
  - siehe Übungsbeispiel
- …

Mechanismen – ein Überblick

- Mutex, Lock
  - lock $\rightarrow$ owner (Besitzer)
  - unlock $\rightarrow$ nur vom gleichen Thread (Besitzer)!!
- Bedingungsvariable
  - Datenstruktur, die Threads repräsentiert auf die gewartet wird
  - Operationen: `notify_one`, `notify_all`, `wait`, `wait_for`, `wait_until`
- Monitor
- Semaphor
- Read-Write Lock
- atomare Variable
- Promise & Future, Task

Monitor

- klassischerweise eine Sammlung von Prozeduren und Datenstrukturen,
  - die als Einheit gruppiert sind.
- Prozesse können die Prozeduren eines Monitor aufrufen,
  - aber nicht auf die internen Datenstrukturen zugreifen.
- Es können nicht zwei Prozesse gleichzeitig in einem Monitor aktiv sein!
- Bedingungsvariablen (condition variables) zusammen mit zwei Operationen WAIT und NOTIFY (oder auch SIGNAL genannt)
- $\rightarrow$ Java, C#

Semaphor

- …zur Verwaltung begrenzter Ressourcen
  - verwaltet diese nicht selber, sondern nur Anzahl
    - z.B. Karten mit freier Platzwahl im Kino
    - z.B. 10 Lizenzen für ein SW Produkt
  - hat *keinen* Besitzer
- …ist ein Zähler,
  - dessen Wert immer ≥ 0 ist
  - Zähler: (atomar) inkrementiert bzw. dekrementiert
    - inkrementieren: traditionell … ”P”, meist `release`
    - dekrementieren: traditionell … ”V”, meist `acquire`
  - Dekrementieren nur, wenn Zähler > 0, ansonsten blockierende Operation $\rightarrow$ bis anderer Thread inkrementiert
- Nicht in C++, aber leicht mittels `mutex` und `condition_variable` zu implementieren!

Semaphor – Serialisierung

```
Semaphore sem{};
```

```
void a() {
    opa1();
    sem.release();
}
```

```
void b() {
    sem.acquire();
    opb1();
}
```

Semaphor – Rendevous

Erweiterung der Serialisierung, sodass diese in beide Richtungen funktioniert:

```
void a() {
    opa1();
    opa2();
}
```

```
void b() {
    opb1();
    opb2();
}
```

Semaphor – Rendevous – 2

```
Semaphore a_arrived{};
Semaphore b_arrived{};
```

```
void a() {
  opa1();
  a_arrived.release();
  b_arrived.acquire();
  opa2();
}
```

```
void b() {
  opb1();
  b_arrived.release();
  a_arrived.acquire();
  opb2();
}
```

Semaphor – Rendevous – 3

Achtung: Gefahr eines Deadlocks!

```
Semaphore a_arrived{};
Semaphore b_arrived{};
```

```
void a() {
  opa1();
  b_arrived.acquire();
  a_arrived.release();
  opa2();
}
```

```
void b() {
  opb1();
  a_arrived.acquire();
  b_arrived.release();
  opb2();
}
```

Semaphor – Mutex

```
Semaphore mtx{1};  // # max. Threads
cnt = 0;
```

```
void a() {
    mtx.acquire();
    cnt += 1;
    mtx.release();
}
```

```
void b() {
    mtx.acquire();
    cnt += 1;
    mtx.release();
}
```

Semaphor – Latch

```
Semaphore mtx{1};
Semaphore latch{};
cnt = 0;

// each thread
opbefore();

mtx.acquire();
cnt += 1;
if (cnt == n) latch.release();
mtx.release();

latch.acquire();
latch.release();

opafter();
```

Semaphor – Barrier

- Latch kann nicht mehr verwendet werden!
- Barrier ist ein Latch, das wiederverwendet werden kann
- Achtung: Begriffe!
  - Ein Latch wird oft als Barrier bezeichnet
  - Ein Barrier wird oft als Cyclic Barrier bezeichnet

Semaphor – Producer/Consumer

```
Semaphore full_cnt{};
Semaphore empty_cnt{4}; // replace n appropriately!
Semaphore mtx{1};

void put(WorkPacket p) {
    empty_cnt.acquire();
    mtx.acquire();
    // add p to queue
    mtx.release();
    full_cnt.release();
}
```

Semaphor – Producer/Consumer – 2

```
WorkPacket take() {
    full_cnt.acquire();
    mtx.acquire();
    // get p from queue
    mtx.release();
    empty_cnt.release();
    return p;
}
```

- `empty_cnt` und `full_cnt` spiegeln nicht die tatsächliche Anzahl an leeren und vollen Plätzen wieder (Zeit!)
- $\texttt{empty\_cnt} + \texttt{full\_cnt} \leq \texttt{n}$

Read-Write Lock

- Readers/Writers Problem
- Einsatz, wenn
  - Zugriffe in lesend und schreibend unterteilbar
  - mehr lesende als schreibende Zugriffe
  - der Overhead akzeptabel ist
- In C++ 14 mittels `shared_lock` / `unique_lock` realisierbar

Read-Write Lock – 2

```cpp
#include <iostream>  // rwlock.cpp
#include <shared_mutex>  // since C++14!
#include <thread>
#include <random>
using namespace std;
using namespace std::chrono;
// ATTN: only shared_timed_mutex until C++14!
shared_mutex mtx;
```

Read-Write Lock – 3

```
void reader(string name) {
    random_device rd;
    mt19937 gen{rd()};
    uniform_int_distribution<> dis{100, 500};
    while (true) {
        this_thread::sleep_for(
          milliseconds{dis(gen)});
        shared_lock<shared_mutex> sl{mtx};
        cout << name << ": enters" << endl;
        this_thread::sleep_for(milliseconds{100});
        cout << name << ": leaves" << endl;
    }
}
```

Read-Write Lock – 4

```
void writer(string name) {
    random_device rd;
    mt19937 gen{rd()};
    uniform_int_distribution<> dis{0, 100};
    while (true) {
        this_thread::sleep_for(
          milliseconds{dis(gen)});
        unique_lock<shared_mutex> ul{mtx};
        cout << name << ": enters" << endl;
        this_thread::sleep_for(milliseconds{1000});
        cout << name << ": leaves" << endl;
    }
}
```

Read-Write Lock – 5

```
int main() {
    thread rdr1{reader, ”r1”};
    thread rdr2{reader, ”r2”};
    thread rdr3{reader, ”r3”};
    thread wtr{writer, ”w1”};
    rdr1.join();
    rdr2.join();
    rdr3.join();
    wtr.join();
}
```

Read-Write Lock – 6

Beispielausgabe:

```
w1: enters
w1: leaves
r1: entersr2: enters
r3: enters

r1: leaves
r3: leaves
```

Atomare Variable

```
#include <iostream>
#include <thread>
#include <atomic>
using namespace std;
struct AtomicCounter {
  atomic<int> value{};
  void incr(){ ++value; }
  void decr(){ --value; }
  int get(){ return value.load(); } };

int main() {  AtomicCounter c;
  thread t1{[&c](){ c.incr(); cout << c.get(); }};
  thread t2{[&c](){ c.incr(); cout << c.get(); }};
  t1.join();  t2.join(); }
```

```
21
```