Datenstrukturen und Algorithmen

Übung 14 - Nachbesprechung Übung 13

FS 2021

Programm von heute

1. Feedback letzte Übung

Aufgabe 13.2: Race conditions

- `Item` Funktionen thread-safe machen.
- Einfacher Ansatz, lock zu Beginn jeder Funktion holen, am Ende der Funktion freigeben.

Ratings

```
class Item {
private:
  int rating_sum = 0;
  int rating_count = 0;
  std::recursive_mutex mtx; // re-entrant lock for out_rating
public:
  Item() {};

  /* Returns average rating. 0 if no rating occured */
  double get_rating() {
    // minimal requirement: do not forget the lock
    std::lock_guard<std::recursive_mutex> lock(mtx);
    if(rating_count == 0) return 0.0; // some forgot this
    return (double)rating_sum / rating_count;
  }
```

Ratings

```
void add_rating(int stars){
  assert(1 <= stars && stars <= 5);
  std::lock_guard<std::recursive_mutex> lock(mtx);
  // some put the computation of the rating here,
  // which is quite clever
  rating_sum += stars;
  rating_count++;
}
```

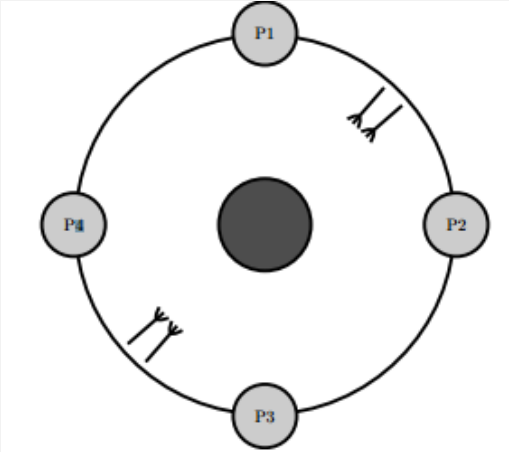
Ratings

```
 // when you do not protect this, you might run into two kind of problems:
 // 1.) Inconsistent result
 //     when call to add_rating between rating_count and get_rating
 // 2.) scrumbled output when threads call out_rating in parallel
 void out_rating(){
    std::lock_guard<std::recursive_mutex> lock(mtx); // required!
    std::cout << "ratings:" << rating_count << ", ";
    std::cout << "score:" << get_rating() << "\n";
  }
};
```

13.3. Dining Philosophers

- Um Deadlock zu verhindern, zirkuläre Abhängigkeit brechen. Wie letztes Mal diskutiert.
- Max/Min Anzahl Philosophen die gleichzeitig essen?
- Es ist möglich dass nur ein Philosph isst.

Gabeln bündeln! Dann können immer zwei essen.



13.4. Bridge

Sicherstellen, dass maximal 3 Autos oder ein LKW gleichzeitig auf der Brücke ist

Verwende condition variable und einen Zähler

Bridge

```
class Bridge {
    public:
       std::mutex mtx;
       std::condition_variable cv;

       int car_count = 0;

       void check_bridge(){
            if(car_count > 3){
                std::cout << "Bridge collapsed!" << std::endl;
                exit(0);
            }
       }
```

Bridge

```
void enter_car(){
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [&]{return car_count < 3;});
    car_count++;
    check_bridge();
}

void leave_car(){
    std::lock_guard<std::mutex> lock(mtx);
    car_count--;
    cv.notify_all();
}
```

Bridge

```
    void enter_truck(){
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [&]{return car_count == 0;});
        car_count += 3;
        check_bridge();
    }

    void leave_truck(){
        std::lock_guard<std::mutex> lock(mtx);
        car_count -= 3;
        cv.notify_all();
    }
};
```

Problem mit diesem Ansatz?

Was passiert, wenn sich vor der Brücke ein Stau von Autos und LKWs bildet? (Annahme: separate Spuren für Autos und LKWs)

Problem mit diesem Ansatz?

Was passiert, wenn sich vor der Brücke ein Stau von Autos und LKWs bildet? (Annahme: separate Spuren für Autos und LKWs)

LKWs kommen nicht vorwärts, da permanent Autos nachkommen.

Problem mit diesem Ansatz?

Was passiert, wenn sich vor der Brücke ein Stau von Autos und LKWs bildet? (Annahme: separate Spuren für Autos und LKWs)

LKWs kommen nicht vorwärts, da permanent Autos nachkommen.

Lösung?

Problem mit diesem Ansatz?

Was passiert, wenn sich vor der Brücke ein Stau von Autos und LKWs bildet? (Annahme: separate Spuren für Autos und LKWs)

LKWs kommen nicht vorwärts, da permanent Autos nachkommen.

Lösung? **Convoy-Verbot:** Lasse Autos nur zu, wenn kein LKW wartet und weniger als 3 Autos (und kein LKW) auf der Brücke oder wenn 0 Autos auf der Brücke.

Dann reduziert sich die Fairness auf die Zuteilung des Laufzeitsystems.

Fairness

```
class Bridge {
  std::mutex mtx;
  std::condition_variable cv;

  int car_count = 0; // count car equivalence
  int trucks_waiting = 0; // count trucks waiting
public:
```

Fairness

```
void enter_car(){
  std::unique_lock<std::mutex> lock(mtx);
  cv.wait(lock, [&]{
      return (car_count < 3)
            && (trucks_waiting == 0 || car_count == 0);}
  );
  car_count++;
  check_bridge();
}

void leave_car(){
  std::lock_guard<std::mutex> lock(mtx);
  car_count--;
  cv.notify_all();
}
```

Fairness

```
  void enter_truck(){
    std::unique_lock<std::mutex> lock(mtx);
    trucks_waiting++;
    cv.wait(lock, [&]{return car_count = 0;});
    trucks_waiting--;
    car_count += 3;
    check_bridge();
  }

  void leave_truck(){
    std::lock_guard<std::mutex> lock(mtx);
    car_count -= 3;
    cv.notify_all();
  }
};
```