ETH zürich

# Exercise Session 14

Data Structures and Algorithms, D-MATH, ETH Zurich

# Program of today

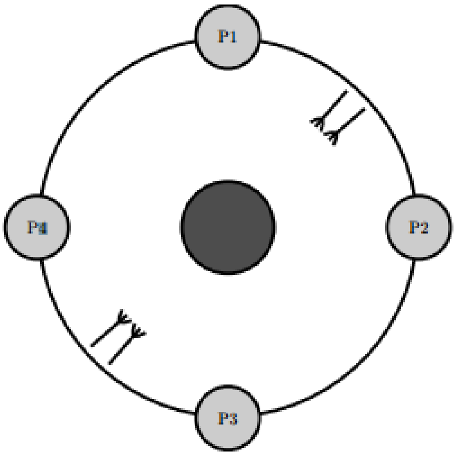
Feedback of last exercise

# 1. Feedback of last exercise

# Dining Philosophers

- To avoid deadlocks, break cyclic dependency. As discussed last time.
- Max/Min numbers of philosophers eating concurrently?
- It's possible that only one philosopher eats.

Bundle forks! Then always two can eat.



# Race conditions

- Make functions of `Item` class thread safe.
- Simple approach: Get lock at beginning of function, release at the end.

# 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";
  }
};
```

# Bridge

Ensure that at most three cars or one truck is on the bridge

Use condition variable and a counter

# 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 with this Approach?

What happens if there are cars and trucks waiting at the bridge?

# Problem with this Approach?

What happens if there are cars and trucks waiting at the bridge?

The trucks do not make progress because cars.

# Problem with this Approach?

What happens if there are cars and trucks waiting at the bridge?

The trucks do not make progress because cars.

Solution?

# Problem with this Approach?

What happens if there are cars and trucks waiting at the bridge?

The trucks do not make progress because cars.

Solution? **Prohibt convoys:** Admit cars only if there is no truck waiting and less than 3 cars (and no truck) on the bridge or there are no cars on the bridge.

The fairness is reduced to the fairness of scheduling by the runtime system.

# 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();
  }
};
```