Datenstrukturen und Algorithmen

Exercise 13

FS 2019

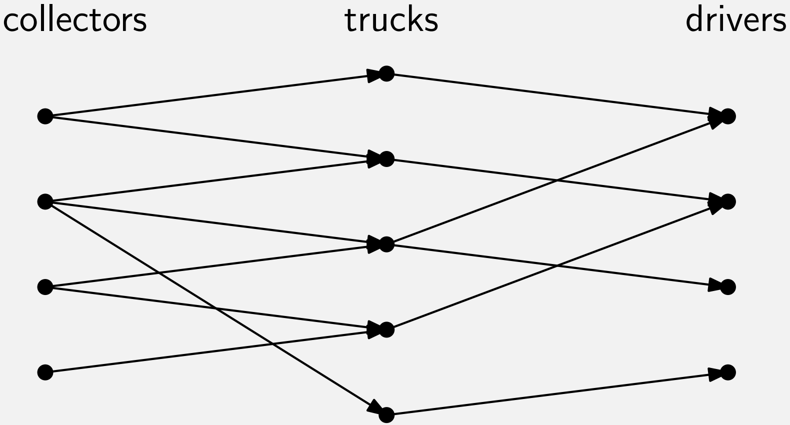
Program of today

1. Feedback of last exercise
2. Repetition theory
3. Next Exercise

1. Feedback of last exercise

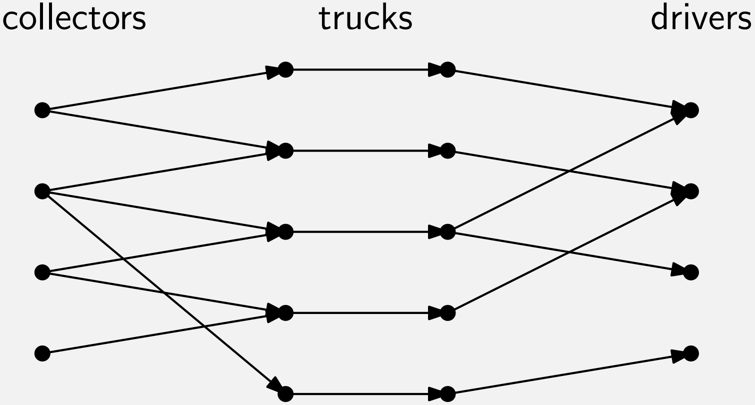
Exercise Applying Maximum Flow

- We have collectors, drivers, and trucks



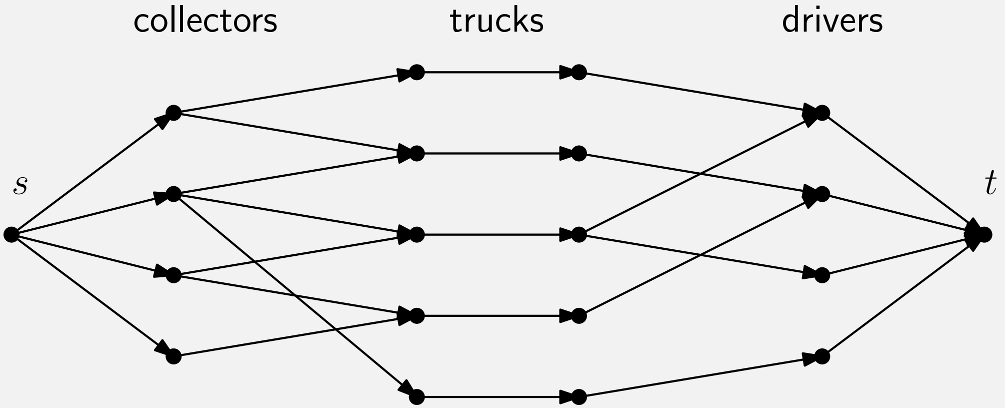
Exercise Applying Maximum Flow

- We have collectors, drivers, and trucks



Exercise Applying Maximum Flow

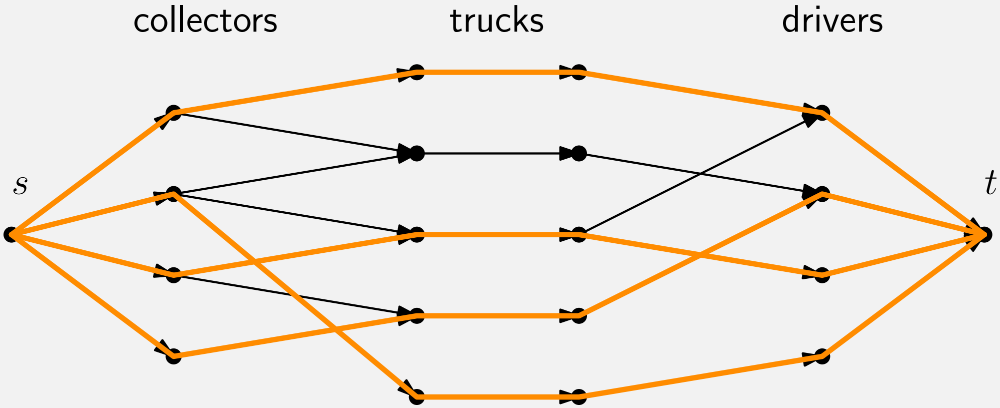
- We have collectors, drivers, and trucks



all edges: capacity 1

Exercise Applying Maximum Flow

- We have collectors, drivers, and trucks



all edges: capacity 1

Exercise: Sum of a vector

```
void sum_par( Iterator beg, Iterator end, int& result ) {
  const int nThreads = std::thread::hardware_concurrency();
  std::vector<std::thread> myThreads;
  std::vector<int> sums( nThreads, 0 );
  const int partSize = (end-beg)/nThreads;

  for( int i=0; i<nThreads-1; ++i ){
    myThreads.emplace_back(
      std::thread(sum_ser, beg, beg + partSize, std::ref(sums[i])));
    beg += partSize;
  }
  // ...
  for( auto& t:myThreads ) t.join();
  sum_ser( sums.begin(), sums.end(), result );
}
```

Exercise: Sum of a vector

```
void sum_ser(
    Iterator from,
    Iterator to,
    int& result ) {

  int local = 0;
  for( ;from != to; ++from )
    local += *from;
  result = local;
}
```

```
void sum_ser(
    Iterator from,
    Iterator to,
    int& result ) {

  result = 0;
  for( ;from != to; ++from )
    result += *from;
}
```

Exercise: Sum of a vector

```
void sum_ser(
    Iterator from,
    Iterator to,
    int& result ) {

  int local = 0;
  for( ;from != to; ++from )
    local += *from;
  result = local;
}
```

```
void sum_ser(
    Iterator from,
    Iterator to,
    int& result ) {

  result = 0;
  for( ;from != to; ++from )
    result += *from;
}
```

Difference?

Exercise: Sum of a vector

```
void sum_ser(
    Iterator from,
    Iterator to,
    int& result ) {

  int local = 0;
  for( ;from != to; ++from )
    local += *from;
  result = local;
}
```

```
void sum_ser(
    Iterator from,
    Iterator to,
    int& result ) {

  result = 0;
  for( ;from != to; ++from )
    result += *from;
}
```

Difference?

execution time: 0.468879 ms

execution time: 0.944031 ms

Exercise: Sum of a vector – False Sharing!

```
void sum_ser(
    Iterator from,
    Iterator to,
    int& result ) {

  int local = 0;
  for( ;from != to; ++from )
    local += *from;
  result = local;
}
```

```
void sum_ser(
    Iterator from,
    Iterator to,
    int& result ) {

  result = 0;
  for( ;from != to; ++from )
    result += *from;
}
```

Difference?

execution time: 0.468879 ms

execution time: 0.944031 ms

Exercise: Mergesort (2-threads)

```
void mergesort_par( std::vector<int> & v ) {
  int n = v.size();
  int partSize = n / 2;

  std::thread t1( mergesort, std::ref(v), 0, partSize-1 );
  std::thread t2( mergesort, std::ref(v), partSize, n-1 );
  t1.join();
  t2.join();
  merge( v, 0, partSize-1, n-1 );
}
```

analogously with n threads

Exercise: Mergesort Recursively

```
void mergesort_par(std::vector<int> & v, int cutoff, int l, int r) {
  if (r-l < cutoff){ // sequential base case
    mergesort( v, l, r );
  } else {
    int m = ( l+r )/2 ;
    std::thread t (mergesort_par,std::ref(v),cutoff,l,m);
    mergesort_par(v,cutoff,m+1,r); // avoid forking another thread
    t.join();
    merge(v,l,m,r);
  }
}
```

2. Repetition theory

Race Conditions

Data Race (low-level Race-Conditions) Erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads, e.g. Simultaneous read/write or write/write of the same memory location

Bad Interleaving (High Level Race Condition) Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm, even if that makes use of otherwise well synchronized resources.

Memory Models

When and if effects of memory operations become visible for threads, depends on hardware, runtime system and programming language.

A *memory model* (e.g. that of C++) provides minimal guarantees for the effect of memory operations

- leaving open possibilities for optimisation
- containing guidelines for writing thread-safe programs

For instance, C++ provides *guarantees when synchronisation with a mutex* is used.

Counter Problem

```
std::vector<std::thread> tv(10);
int counter {0};
for (auto & t:tv)
  t = std::thread([&]{
    for (int i =0; i<100000; ++i){counter++;} // race!!
  });
for (auto & t:tv)
  t.join();
std::cout << "count= "<< counter << std::endl;
```

Counter Solution 1

```
std::vector<std::thread> tv(10);
std::mutex lock;
int counter {0};
for (auto & t:tv)
  t = std::thread([&]{
  for (int i =0; i<100000; ++i){
    mutex.lock(); counter++; mutex.unlock(); // synchronized!
  }});
for (auto & t:tv)
  t.join();
std::cout << "count= "<< counter << std::endl;
```

Counter Solution II

```
std::vector<std::thread> tv(10);
std::atomic<int> counter {0};
for (auto & t:tv)
  t = std::thread([&]{
    for (int i =0; i<100000; ++i){counter++;} // atomic!!
  });
for (auto & t:tv)
  t.join();
std::cout << "count= "<< counter << std::endl;
```

Quiz:What's wrong with this code?

```
void exchangeSecret(Person & a, Person & b) {
  a.getMutex()->lock();
  b.getMutex()->lock();
  Secret s = a.getSecret();
  b.setSecret(s);
  a.getMutex()->unlock();
  b.getMutex()->unlock()
}
```

Deadlock

Thread 1:

```
exchangeSecret(p1, p2);
```

Thread 2:

```
exchangeSecret(p2, p1);
```

Deadlock

Thread 1:

```
exchangeSecret(p1, p2);
```

Thread 2:

```
exchangeSecret(p2, p1);
```

How to resolve?

Possible Solution

```
void exchangeSecret(Person & a, Person & b) {
  std::mutex* first;
  std::mutex* second;
  if (a.name < b.name){
    first = a.getMutex(); second = b.getMutex();
  } else {
    first = b.getMutex(); second = a.getMutex();
  }
  first->lock();
  second->lock();
  Secret s = a.getSecret();
  b.setSecret(s);
  first->unlock();
  second->unlock();
}
```

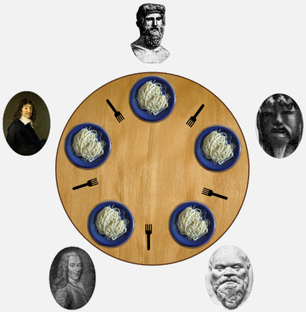
Deadlocks and Races

- Not easy to spot
- Hard to debug
- Might happen only very rarely
- Testing usually not good enough
- Reasoning about code is required

Lesson learned: Need to be careful when programming with locks!

3. Next Exercise

Dining Philosophers



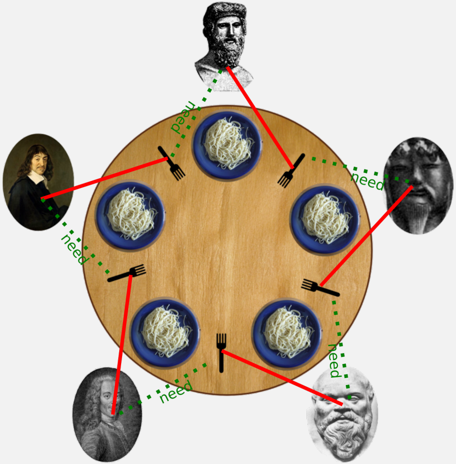
- Philosophers only think and eat. Each needs two forks to eat.
- Philosophers = threads, forks = locks.

Dining Philosophers - pseudocode

```
while(true) {
    think();
    acquire_fork_on_left_side();
    acquire_fork_on_right_side();
    eat();
    release_fork_on_right_side();
    release_fork_on_left_side();
}
```

- Problems with this code?

Dining Philosophers - deadlock



- Solutions?

Dining Philosophers

- Resolve cyclic dependency
- For instance: Philosoph five takes first the **right** fork.
- General solution: Define lock order. Then, always lock in that order.

Locking Datastructures

Coarse-grained Locking: Few locks (one typically) per object. Every object operation acquires the lock first.

Fine-grained Locking: Multiple locks, that protect a less. Usually one per element

Coarse-grained Locking - Example

```
class List {
  std::mutex m;
public:
  void push_back(int amount) {
    std::lock_guard<std::mutex> guard(m);
    ...
    };

    void pop_front() {
    std::lock_guard<std::mutex> guard(m);
    ...
    };
};
```

Fine-grained Locking - Linked List

Consider a **single linked list**. How to do fine grained locking?

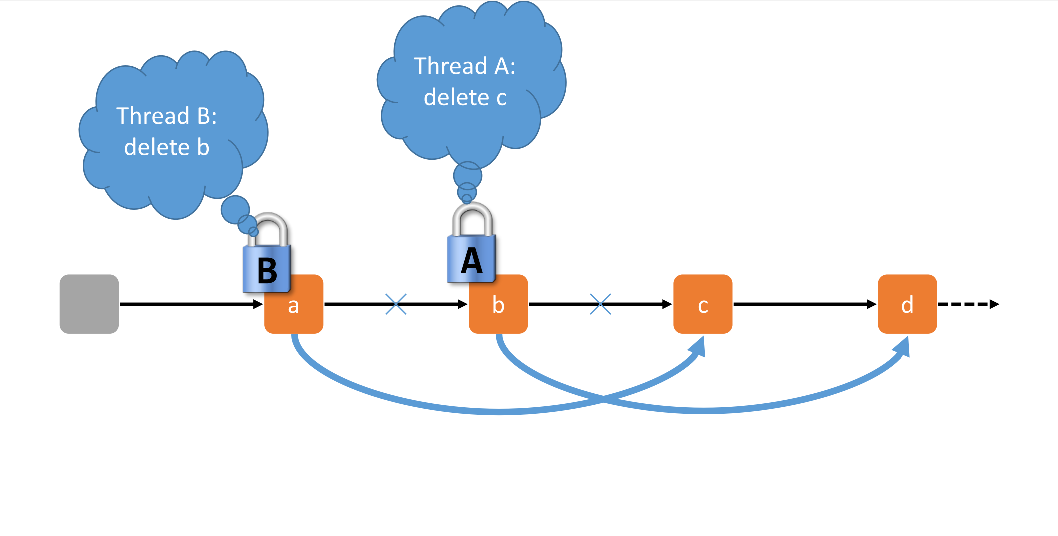
Fine-grained Locking - Linked List

Consider a **single linked list**. How to do fine grained locking?

First idea: One lock per list item. When changing the element, the lock must be held. (For instance when changing the next pointer due to an insertion)

But is this enough?

Fine-grained locking - Linked List

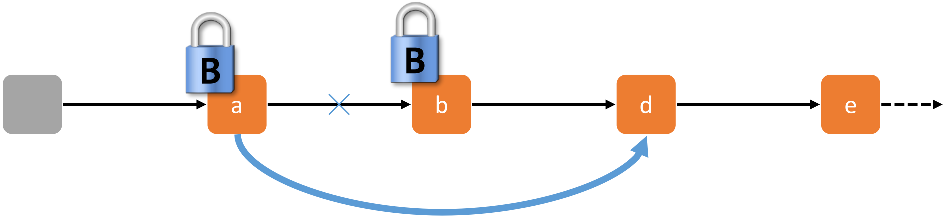


Fine-grained locking - Linked List

C not deleted ☹

Fine-grained locking - Hand-over-hand locking

Solution? Also lock the next element.



Fine-grained locking - Linked List

Is locking necessary when traversing?

Fine-grained locking - Linked List

Is locking necessary when traversing?

Yes, the element we are currently looking at can be deleted.

Fine-grained locking - Linked List

Is locking necessary when traversing?

Yes, the element we are currently looking at can be deleted.

Lock order?

Fine-grained locking - Linked List

Is locking necessary when traversing?

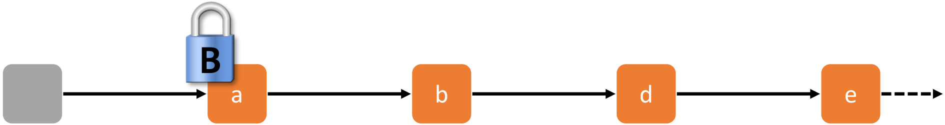
Yes, the element we are currently looking at can be deleted.

Lock order?

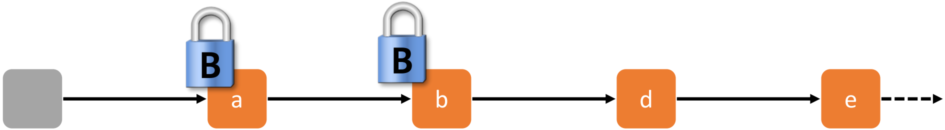
Acquire next lock before releasing current one. This is called *hand-over-hand* locking

Implementation hint: Don't use `lock_guard`, but call directly `lock` and `unlock`.

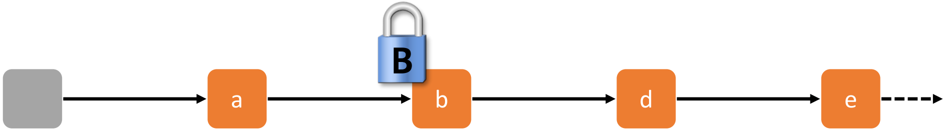
Fine-grained locking - Linked List



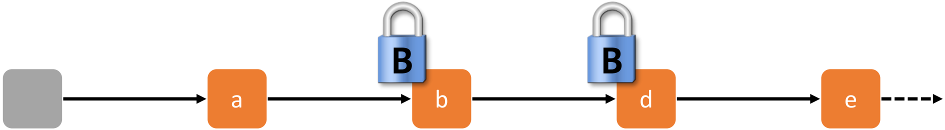
Fine-grained locking - Linked List



Fine-grained locking - Linked List



Fine-grained locking - Linked List



Condition variables

Condition variables allow a thread to wait efficiently on a specific condition.

Once the condition has changed (or could have been changed), the changing thread notifies the waiting one(s).

Condition Variables

```
class Buffer {
...
public:
    void put(int x){
        guard g(m);
        buf.push(x);
        cond.notify_one();
    }
    int get(){
        guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        int x = buf.front(); buf.pop();
        return x;
    }
};
```

Questions?