Datenstrukturen und Algorithmen

Übung 13

FS 2020

Programm von heute

1. Feedback letzte Übung
2. Wiederholung Theorie
3. Nächste Übung

1. Feedback letzte Übung

Aufgabe: Summe eines Vektors

```
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 );
}
```

Aufgabe: Summe eines Vektors

```
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;
}
```

Aufgabe: Summe eines Vektors

```
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?

Aufgabe: Summe eines Vektors

```
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

Aufgabe: Summe eines Vektors – 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

Aufgabe: 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 );
}
```

Analog mit n threads

Aufgabe: Mergesort Rekursiv

```
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. Wiederholung Theorie

Race Conditions (Wettlaufsituationen)

Data Race (low-level Race-Conditions) Fehlerhaftes Programmverhalten verursacht durch ungenügend synchronisierten Zugriff zu einer gemeinsam genutzten Resource, z.B. gleichzeitiges Lesen/Schreiben oder Schreiben/Schreiben zum gleichen Speicherbereich.

Bad Interleaving (High Level Race Condition) Fehlerhaftes Programmverhalten verursacht durch eine unglückliche Ausführungsreihenfolge eines Algorithmus mit mehreren Threads, selbst dann wenn die gemeinsam genutzten Resourcen anderweitig gut synchronisiert sind.

Speichermodelle

Wann und ob Effekte von Speicheroperationen für Threads sichtbar werden, hängt also von Hardware, Laufzeitsystem und der Programmiersprache ab.

Ein *Speichermodell* (z.B. das von C++) gibt Minimalgarantien für den Effekt von Speicheroperationen.

- Lässt Möglichkeiten zur Optimierung offen
- Enthält Anleitungen zum Schreiben Thread-sicherer Programme

C++ gibt zum Beispiel *Garantien, wenn Synchronisation mit einer Mutex verwendet* wird.

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 Lösung 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 Lösung 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:Was ist hier falsch?

```
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 (Verklemmung)

Thread 1:

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

Thread 2:

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

Deadlock (Verklemmung)

Thread 1:
```
exchangeSecret(p1, p2);
```

Thread 2:
```
exchangeSecret(p2, p1);
```

Was tun?

Mögliche Lösung

```
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();
}
```

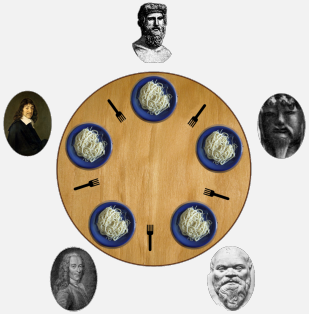
Deadlocks und Races

- Nicht einfach zu sehen
- Schwierig zu debuggen
- Treten u.U. selten auf
- Testing genügt nicht
- Eigentlich muss man die Korrektheit des Codes formal beweisen

Vorsicht und Sorgfalt ist gefragt beim Programmieren mit Locks!

3. Nächste Übung

Dining Philosophers



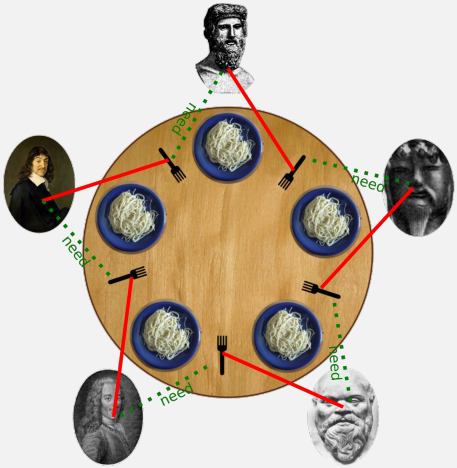
- Philosophen denken und essen abwechslungsweise. Zum Essen brauchen sie zwei Gabeln.
- Philosoph = Thread, Gabel = Lock.

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();
}
```

- Problem mit diesem Program?

Dining Philosophers - Deadlock



- Lösung?

Dining Philosophers

- Zyklische Abhängigkeit brechen
- Beispielsweise: Philosoph fünf nimmt erste **rechte** Gabel.
- Allgemeine Möglichkeit: Lock Ordnung definieren. Dann immer in dieser Reihenfolge locken.

Lock Granularität

Grobgranulares Locking: Wenige Locks (typischerweise eines) pro Objekt, das jede Objekt-Operation zuerst holt.

Feingranulares Locking: Mehrere locks, die kleinere Bereiche schützen. Üblicherweise eines pro Element.

Coarse-grained Locking - Beispiel

```
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 - Verkettete Liste

Betrachte eine **einfach verkettete Liste**, feingranulares Locking?

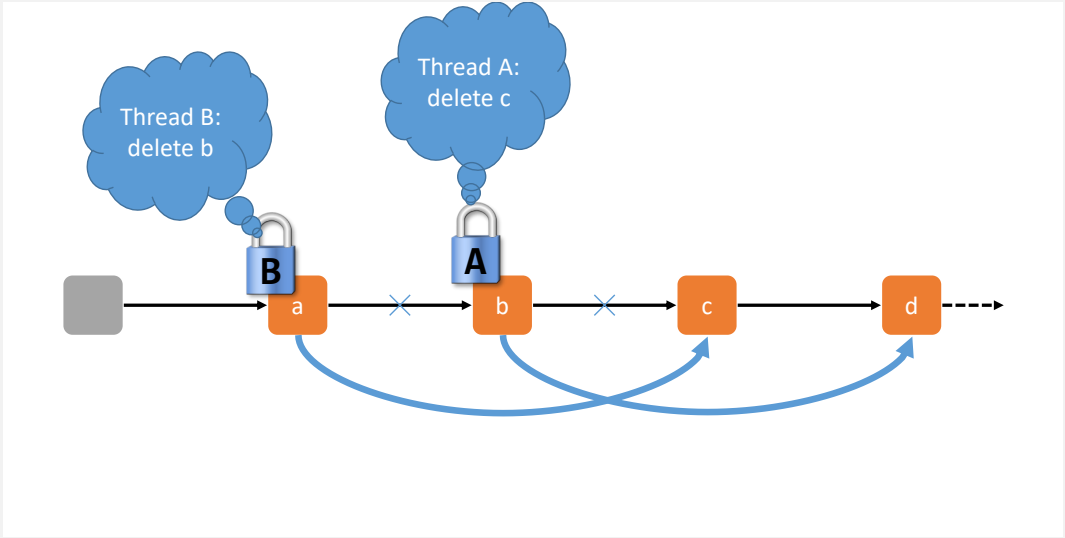
Fine-grained Locking - Verkettete Liste

Betrachte eine **einfach verkettete Liste**, feingranulares Locking?

Erste Idee: Ein Lock pro Listenelement. Beim Ändern des Listenelements muss das Lock gehalten werden. (z.b. Ändern des next pointers wegen einer Einfügeoperation.)

Aber ist das genug?

Fine-grained locking - Linked List

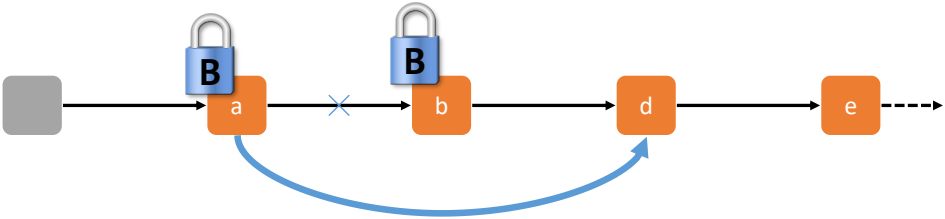


Fine-grained locking - Linked List

C not deleted ☹

Fine-grained locking - Hand-over-hand locking

Lösung? Auch das nächste Element sichern.



Fine-grained locking - Linked List

Ist auch beim Traversieren ein Lock benötigt?

Fine-grained locking - Linked List

Ist auch beim Traversieren ein Lock benötigt?

Ja, sonst kann das Element gelöscht werden welches gerade betrachtet wird.

Fine-grained locking - Linked List

Ist auch beim Traversieren ein Lock benötigt?

Ja, sonst kann das Element gelöscht werden welches gerade betrachtet wird.

Lock Reihenfolge?

Fine-grained locking - Linked List

Ist auch beim Traversieren ein Lock benötigt?

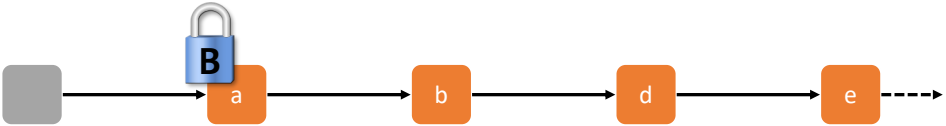
Ja, sonst kann das Element gelöscht werden welches gerade betrachtet wird.

Lock Reihenfolge?

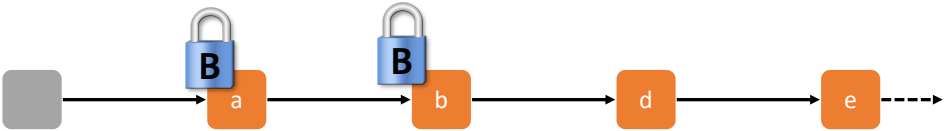
Immer erst das nächste Lock holen vor dem Freigeben. Sogenanntes *hand-over-hand* Locking

Hinweis zur Implementierung: Kein `lock_guard` verwenden, sondern direkt `lock` und `unlock`.

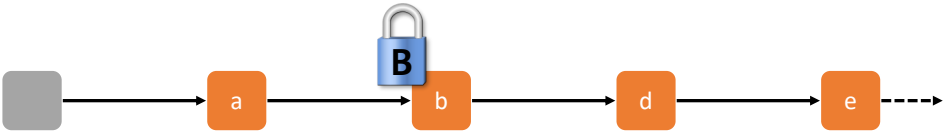
Fine-grained locking - Linked List



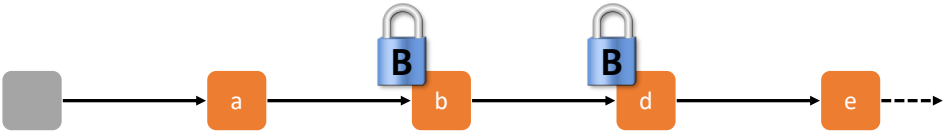
Fine-grained locking - Linked List



Fine-grained locking - Linked List



Fine-grained locking - Linked List



Bedingungsvariabeln

Bedingungsvariabeln (*condition variables*) erlauben einem Thread effizient auf eine gewisse Bedingung zu warten.

Sobald sich die Bedingung geändert hat (oder geändert haben könnte), benachricht der verursachende Thread den/die wartenden Threads.

Bedingungsvariablen

```
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;
    }
};
```

Fragen?