Thread safe collections

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 11

Concurrency and collections

- Synchronize access to bank account array to ensure consistent updates

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Concurrency and collections

- Synchronize access to bank account array to ensure consistent updates
- Noninterfering updates can safely happen in parallel
  - Updates to different accounts, `accounts[i]` and `accounts[j]`

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Concurrency and collections

- Synchronize access to bank account array to ensure consistent updates
- Noninterfering updates can safely happen in parallel
  - Updates to different accounts, `accounts[i]` and `accounts[j]`
- Insistence on sequential access affects performance

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Concurrency and collections

- Synchronize access to bank account array to ensure consistent updates
- Noninterfering updates can safely happen in parallel
  - Updates to different accounts, `accounts[i]` and `accounts[j]`
- Insistence on sequential access affects performance
- Can we implement collections to allow such concurrent updates in a safe manner — make them thread safe?

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment `accounts[i]`, neither update is lost

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment `accounts[i]`, neither update is lost
- Individual updates are implemented in an atomic manner

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment `accounts[i]`, neither update is lost
- Individual updates are implemented in an atomic manner
- Does not say anything about sequences of updates

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment `accounts[i]`, neither update is lost
- Individual updates are implemented in an atomic manner
- Does not say anything about sequences of updates
- Formally, linearizability

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment `accounts[i]`, neither update is lost
- Individual updates are implemented in an atomic manner
- Does not say anything about sequences of updates
- Formally, linearizability
- Contrast with serializability in databases, where transactions (sequences of updates) appear atomic

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restrict access to nodes on either side of insert/delete

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restrict access to nodes on either side of insert/delete
- Java provides built-in collection types that are thread safe
  - `ConcurrentMap` interface, implemented as `ConcurrentHashMap`
  - `BlockingQueue`, `ConcurrentSkipList`, ...
  - Appropriate low level locking is done automatically to ensure consistent local updates

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restrict access to nodes on either side of insert/delete
- Java provides built-in collection types that are thread safe
  - `ConcurrentMap` interface, implemented as `ConcurrentHashMap`
  - `BlockingQueue`, `ConcurrentSkipList`, ...
  - Appropriate low level locking is done automatically to ensure consistent local updates
- Remember that these only guarantee atomicity of individual updates

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restrict access to nodes on either side of insert/delete
- Java provides built-in collection types that are thread safe
  - `ConcurrentMap` interface, implemented as `ConcurrentHashMap`
  - `BlockingQueue`, `ConcurrentSkipList`, ...
  - Appropriate low level locking is done automatically to ensure consistent local updates
- Remember that these only guarantee atomicity of individual updates
- Sequences of updates (transfer from one account to another) still need to be manually synchronized to work properly

Usings thread safe queues for synchronization

- Use a thread safe queue for simpler synchronization of shared objects

Usings thread safe queues for synchronization

- Use a thread safe queue for simpler synchronization of shared objects
- Producer–Consumer system
  - Producer threads insert items into the queue
  - Consumer threads retrieve them.

Usings thread safe queues for synchronization

- Use a thread safe queue for simpler synchronization of shared objects
- Producer–Consumer system
  - Producer threads insert items into the queue
  - Consumer threads retrieve them.
- Bank account example
  - Transfer threads insert transfer instructions into shared queue
  - Update thread processes instructions from the queue, modifies bank accounts
  - Only the update thread modifies the data structure
  - No synchronization necessary

Usings thread safe queues for synchronization

- Use a thread safe queue for simpler synchronization of shared objects
- Producer–Consumer system
  - Producer threads insert items into the queue
  - Consumer threads retrieve them.
- Bank account example
  - Transfer threads insert transfer instructions into shared queue
  - Update thread processes instructions from the queue, modifies bank accounts
  - Only the update thread modifies the data structure
  - No synchronization necessary
- How does a consumer thread know when to check the queue?

Blocking queues

- Blocking queues block when . . .
  - . . . you try to add an element when the queue is full
  - . . . you try to remove an element when the queue is empty

Blocking queues

- Blocking queues block when . . .
  - . . . you try to add an element when the queue is full
  - . . . you try to remove an element when the queue is empty
- Update thread tries to remove an item to process, waits if nothing is available

Blocking queues

- Blocking queues block when . . .
  - . . . you try to add an element when the queue is full
  - . . . you try to remove an element when the queue is empty
- Update thread tries to remove an item to process, waits if nothing is available
- In general, use blocking queues to coordinate multiple producer and consumer threads
  - Producers write intermediate results into the queue
  - Consumers retrieve these results and make further updates

Blocking queues

- Blocking queues block when ...
  - ... you try to add an element when the queue is full
  - ... you try to remove an element when the queue is empty
- Update thread tries to remove an item to process, waits if nothing is available
- In general, use blocking queues to coordinate multiple producer and consumer threads
  - Producers write intermediate results into the queue
  - Consumers retrieve these results and make further updates
- Blocking automatically balances the workload
  - Producers wait if consumers are slow and the queue fills up
  - Consumers wait if producers are slow to provide items to process

Summary

- When updating collections, locking the entire data structure for individual updates is wasteful
- Sufficient to protect access within a local portion of the structure
  - Ensure that two updates do not overlap
  - Region to protect depends on the type of collection
  - Implement using lower level locks of suitable granularity
- Java provides built-in thread safe collections
- One of these is a blocking queue
  - Use a blocking queue to coordinate producers and consumers
  - Ensure safe access to a shared data structure without explicit synchronization