Race conditions

Madhavan Mukund

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

Programming Concepts using Java

Week 10

Threads and shared variables

- Threads are lightweight processes with shared variables that can run in parallel
- Browser example: download thread and user-interface thread run in parallel
  - Shared boolean variable `terminate` indicates whether download should be interrupted
  - `terminate` is initially false
  - Clicking `Stop` sets it to true
  - Download thread checks the value of this variable periodically and aborts if it is set to true
- Watch out for race conditions
  - Shared variables must be updated consistently

Maintaining data consistency

- `double accounts[100]` describes 100 bank accounts

Maintaining data consistency

- double accounts[100] describes 100 bank accounts
- Two functions that operate on accounts: transfer() and audit()

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

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

Maintaining data consistency

- `double accounts[100]` describes 100 bank accounts
- Two functions that operate on `accounts`: `transfer()` and `audit()`
- What are the possibilities when we execute the following?

```
Thread 1                    Thread 2
...                         ...
status =                    System.out.
  transfer(500.00,7,8);      print(audit());
...                         ...
```

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

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

Maintaining data consistency . . .

- What are the possibilities when we execute the following?

```
Thread 1                    Thread 2
...                         ...
status =                    System.out.
  transfer(500.00,7,8);      print(audit());
...                         ...
```

- `audit()` can report an overall total that is 500 more or less than the actual assets

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

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

Maintaining data consistency ...

- What are the possibilities when we execute the following?

```
Thread 1                   Thread 2
...                        ...
status =                   System.out.
  transfer(500.00,7,8);     print(audit());
...                        ...
```

- audit() can report an overall total that is 500 more or less than the actual assets
  - Depends on how actions of transfer are interleaved with actions of audit

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

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

Maintaining data consistency . . .

- What are the possibilities when we execute the following?

```
Thread 1                  Thread 2
...                       ...
status =                  System.out.
  transfer(500.00,7,8);    print(audit());
...                       ...
```

- audit() can report an overall total that is 500 more or less than the actual assets
  - Depends on how actions of transfer are interleaved with actions of audit
  - Can even report an error if transfer happens atomically

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

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

Atomicity of updates

- Two threads increment a shared variable n

```
Thread 1            Thread 2
...                 ...
m = n;              k = n;
m++;                k++;
n = m;              n = k;
...                 ...
```

Atomicity of updates

- Two threads increment a shared variable `n`

```
Thread 1            Thread 2
...                 ...
m = n;              k = n;
m++;                k++;
n = m;              n = k;
...                 ...
```

- Expect `n` to increase by 2 . . .

Atomicity of updates

- Two threads increment a shared variable n

```
Thread 1            Thread 2
...                 ...
m = n;              k = n;
m++;                k++;
n = m;              n = k;
...                 ...
```

- Expect n to increase by 2 . . .

- . . . but, time-slicing may order execution as follows

```
Thread 1: m = n;
Thread 1: m++;
Thread 2: k = n;   // k gets the original value of n
Thread 2: k++;
Thread 1: n = m;
Thread 2: n = k;   // Same value as that set by Thread 1
```

Race conditions and mutual exclusion

- Race condition — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets

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

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

Race conditions and mutual exclusion

- Race condition — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets
- Avoid this by insisting that `transfer()` and `audit()` do not interleave

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

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

Race conditions and mutual exclusion

- Race condition — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets
- Avoid this by insisting that `transfer()` and `audit()` do not interleave
- Never simultaneously have current control point of one thread within `transfer()` and another thread within `audit()`

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

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

Race conditions and mutual exclusion

- Race condition — concurrent update of shared variables, unpredictable outcome
    - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets
- Avoid this by insisting that `transfer()` and `audit()` do not interleave
- Never simultaneously have current control point of one thread within `transfer()` and another thread within `audit()`
- Mutually exclusive access to critical regions of code

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

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

Summary

- Concurrent update of a shared variable can lead to data inconsistenccy
  - Race condition
- Control behaviour of threads to regulate concurrent updates
  - Critical sections — sections of code where shared variables are updated
  - Mutual exclusion — at most one thread at a time can be in a critical section