Parallel Programming

Johan Östlund

In part based on ” Sophomoric Parallelism and Concurrency” by Dan Grossman

http://www.cs.washington.edu/homes/djg/teachingMaterials/spac/

Sequential execution

- One thing happens at a time
- The program has a given order of execution, and so do accesses to resources (e.g., memory)

- Will the assert in this program ever fail?

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1;
    y = 1;
  }

  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }

}
```

Why sequential programming does not work

- More’s Law: "The number of transistors incorporated in a chip will approximately double every 24 months." - Gordon Moore, Intel Co-Founder
- This forecast has been more or less correct for 40 years
- Chip makers used the increasing space and decreasing distance to make faster chips
- This no longer works as increased clock rates lead to increased power consumption and heat generation
- In the last decade chip makers instead have started using the increased space for multiple cores, running at lower speed
- In order to take advantage of these multiple cores programs must be written differently

What can we do with multiple cores?

- In the coming years regular computers are likely to have 4, 8 or 16 cores
- Run multiple programs at the same time

  We already do that, but with time-slicing

- Do multiple things similtaneously in the same program (this will be our focus)

Requires a different mindset, other algorithms, other data structures

Concurrency vs. Parallelism

- Concurrency and parallelism are often confused
- Concurrency is described by some as ”managing access to shared resources”.
- Parallelism is ”doing several things at the same time”, for efficiency
- Threads are commonly used to achieve both, which is probably why they’re easily confused
- If parallel tasks need access to shared resources, then concurrency needs to be managed

Shared memory

- A sequential program has

  one call stack (with local variables)

  one program counter

  one heap (where objects are stored)

- A concurrent/parallel program has

  many call stacks (with their own local variables)

  many program counters

  one heap (where objects are stored)

- See a problem? We need to coordinate (order) accesses to the shared heap.

So what do we need?

- A way to run multiple things simultaneously, let’s call these things threads
- Ways for threads to share data (we already have that in the shared heap)
- Ways for threads to coordinate (synchronize)

Threads in Java

- In Java there is a class java.lang.Thread
- You can subclass this thread and override the `run()` method
- Doing this and calling `start()` will begin execution in a new thread

Concurrent programming

- Concurrency: ”Correctly and efficiently managing access to shared resources from multiple - possibly simultaneous - clients”
- Concurrency requires coordination, particularly synchronization to avoid incorrect simultaneous access. We need a way to block other threads while we’re using a shared resource
- Example: what if two threads update a bank account at the same time?

Concurrent execution

- We use threads for concurrent execution
- If the number of threads is greater than the number of cores they will not all run in parallel. Threads will run in an interleaved manner
- Threads may still be useful for performance

  Hide latency when reading from a file

  Have the GUI respond while waiting for content from a web server

Sharing (again)

- Different threads might access the same resources in an unpredictable order or even at about the same time
- Program correctness requires that simultaneous access be prevented using synchronization
- Simultaneous access is rare, and hard to provoke

  Makes testing and debugging very difficult

Canonical example: The Bank

- This code is correct for a single-threaded execution
- but not for a multi-threaded execution

```
class BankAccount {
  private int balance = 0;

  int getBalance() { return balance; }

  void setBalance(int x) { balance = x; }

  void withdraw(int amount) {
    int b = getBalance();
    if (amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b - amount);
  }

  ...
  // other operations like deposit, etc.
}
```

Interleaving

- Suppose

  Thread 1 calls `x.withdraw(100)`

  Thread 2 calls `y.withdraw(100)`

- If second call starts before first finishes, we say the calls interleave. Could happen even with one processor since a thread can be pre-empted at any point for time-slicing

- If `x` and `y` refer to different accounts, no problem, but if `x` and `y` refer to the same object, there's trouble ahead

Bad interleavings

- Interleaved `withdraw(100)` calls on the same account (Assume initial `balance == 150`)

Thread 1

```
int b = getBalance();

if(amount > b)
  throw new …;
setBalance(b – amount);
```

Thread 2

```
int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b – amount);
```

Time

"Lost withdraw" – unhappy bank

The non-fix

- You may think that the problem is that we're saving the value on the stack and that reading it again will solve the problem?
- Well, it doesn't
- The balance may still change between the if-statement and the call to `setBalance()`
- This is just moving the problem a little, not solving it

```
class BankAccount {
  private int balance = 0;

  int getBalance() { return balance; }

  void setBalance(int x) { balance = x; }

  void withdraw(int amount) {
    if (amount > getBalance())
      throw new WithdrawTooLargeException();
    // maybe balance changed
    setBalance(getBalance() - amount);
  }

  ...
  // other operations like deposit, etc.
}
```

Assert example revisited

- Remember this example?
- Are threre any bad interleavings?
- No! Right? If `!(b>=a)`, then `a==1` and `b==0`. But if `a==1`, then `a=y` happened after `y=1`. And since programs execute in order, `b=x` happened after `a=y` and `x=1` happened before `y=1`. So by transitivity, `b==1`.
- Well, unfortunately that's not the whole story

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1;
    y = 1;
  }

  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }

}
```

Reordering

- For performance reasons the compiler and hardware often reorder memory operations
- A reordering will never be performed that would affect the result of a single-threaded program
- A bit simplified: a reordering will never be performed that crosses a synchronization boundary, so a data-race free program will not be observably affected by reorderings
- If there are no data-races in your program, then you can forget about reorderings
- Data-races are errors! A program with a data-race is always incorrect.

Mutual exclusion

- We need to make sure that at most one thread can withdraw from an account at the same time (and similar for other account operations, e.g., deposit)
- This is called mutual exclusion: One thread using a resource (here: an account) means all other threads must wait if they want to use that same resource
- The programmer must implement critical sections, i.e., tell the compiler which interleavings are allowed
- We need locks

Locks

- A lock is (usually) an object with three basic operations

  Create a new lock

  Acquire a lock, ”block the current thread if the lock is held, otherwise grab the lock

  Release a lock, ”make the lock not held, if other threads are waiting on the lock give it to exactly one of those threads”

- Locks use special hardware and OS support to ensure that the locking is atomic

Bank example (almost correct)

- Why is this just ”almost” correct?

```
class BankAccount {
  private int balance = 0;
  private Lock lk = new Lock();

  ...

  void withdraw(int amount) {
    lk.acquire(); // may block
    int b = getBalance();
    if (amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
    lk.release();
  }

// deposit would also acquire/release lk
}
```

Bank example (correct)

- We need to release the lock on all exit paths.
- Is there a nice way to do that in Java?

```
class BankAccount {
  private int balance = 0;
  private Lock lk = new Lock();

  ...

  void withdraw(int amount) {
    lk.acquire(); // may block
    int b = getBalance();
    if (amount > b) {
      lk.release();
      throw new WithdrawTooLargeException();
    }
    setBalance(b – amount);
    lk.release();
  }

// deposit would also acquire/release lk
}
```

Bank example (also correct)

- We need to release the lock on all exit paths. We ca use Java's try-finally construct
- the finally-clause is always run, no matter how the try-block is exited

```
class BankAccount {
  private int balance = 0;
  private Lock lk = new Lock();

  ...

  void withdraw(int amount) {
    try {
      lk.acquire(); // may block
      int b = getBalance();
      if (amount > b) {
        throw new WithdrawTooLargeException();
      }
      setBalance(b – amount);
    } finally {
      lk.release();
    }
  }

// deposit would also acquire/release lk
}
```

Java: `synchronized`

- In Java there is a built-in language construct called `synchronized`

```
synchronized (expression) {
  statements
}
```

- Every Java object ”is itself a lock”
- The synchronized statement evaluated the expression and uses the resulting object as lock for the synchonized block
- Java automatically releases the lock when the synchronized block is exited, even if it’s with a return or exception

Bank with `synchronized`

```
class BankAccount {
  private int balance = 0;
  private Object lk = new Object();

  int getBalance() {
    synchronized (lk) { return balance; }
  }
  void setBalance(int x) {
    synchronized (lk) { balance = x; }
  }
  void withdraw(int amount) {
    synchronized (lk) {
      int b = getBalance();
      if (amount > b)
        throw ...
      setBalance(b – amount);
    }
  }
// deposit would also use synchronized(lk)
}
```

Bank with `synchronized` (take 2)

```
class BankAccount {
  private int balance = 0;

  int getBalance() {
    synchronized (this) { return balance; }
  }
  void setBalance(int x) {
    synchronized (this) { balance = x; }
  }
  void withdraw(int amount) {
    synchronized (this) {
      int b = getBalance();
      if (amount > b)
        throw ...
      setBalance(b – amount);
    }
  }
// deposit would also use synchronized(lk)
}
```

Bank with `synchronized` (take 3)

```
class BankAccount {
  private int balance = 0;

  synchronized int getBalance() {
    return balance;
  }
  synchronized void setBalance(int x) {
    balance = x;
  }
  synchronized void withdraw(int amount) {
    int b = getBalance();
    if (amount > b)
      throw ...
    setBalance(b – amount);
  }

// deposit would also use synchronized
}
```

The `synchronized` method modifier is a short-hand for enclosing the entire method body in a `synchonized` block with `this` as lock expression

More Java locks

- The package java.util.concurrent contains other types of locks (and tons of other cool stuff)
- You can use these if you need them, but then you cannot use the `synchronized` construct
- Always make sure you enclose the critical section in a try-finally

Common mistakes

- Locks are very primitive, it’s up to you to implement the critical sections and make sure that locks are released correctly
- Incorrect use: if withdraw and deposit use different locks it won’t work: you must use the same lock for the same resource!
- The size of a critical section has impact on performance

  if a single lock is used to lock all access to all accounts we get no parallelism

  if we use too many locks and too small critical sections, well then we lose atomicity (see above). Also, the risk for deadlocks increases

Deadlock

- Deadlock is when two (or more) threads wait on each other
- Ex:

  Thread 1 grabs lock A
  Thread 2 grabs lock B
  Thread 1 tries to grab lock B
  Thread 2 tries to grab lock A

- These threads will wait forever and never make progress

Summing up

- We have talked about multi-threaded execution and briefly the difference between concurrency and parallelism
- In a multi-threaded program (concurrent or parallel) accesses to shared resources must synchronized

    To prevent bad interleavings (sometimes called high-level races), and data-races (low-level races)

- Data-races are errors, and a program with a data-race is broken
- We use locks (synchronization) for mutual exclusion
- Locks are primitive and difficult to get right

Next time: we'll talk about parallelism and how to take advantage of all those cores.