CSC301

Asynchronous patterns & Object pools

Announcements

- A4
    - Due **March 6, 23:59**
    - Marks out later this week (tentatively)
    - In the `auto-marker` branch of your A4 repo (the original one, not your fork)
    - Please let us know if there is a mistake
- Team deliverable 2 is due **March 9, 23:59**

Multithreading

- Multiple threads of execution running “simultaneously”
  - Well … not exactly simultaneously, but sharing the CPU makes it seem like it
  - Java also supports creating new processes, but threads are much more common
- Multithreading is essential to most modern applications
  - Servers handling multiple simultaneous requests
  - GUI applications (GUI updates and background computations happen on separate threads)
  - Applications using 3rd party, remote API’s
    (applications cannot hang while waiting for network response)

Some Benefits of Multithreading

- Improved resource utilization
- Increased responsiveness
    - Chunks of work can be handed over to worker threads - e.g., servers handling requests, GUI
- Efficient use of multiprocessors
- Improving program structure
    - Simplified code, less need to coordinate concurrent tasks

Multithreading

- Multi-threaded programming is traditionally considered difficult
  - Thread interaction can be difficult
  - Easy to get things wrong
    For example: deadlock, race condition, starvation, etc.
  - Harder to reason about and debug
    (mainly because the order of execution is nondeterministic)
    - Errors difficult to detect, reproduce, and fix

Java & Multithreading

- The API's related to threads (and concurrency) evolved over the years
- Initially, there was just Thread
  - Either extend it (i.e., create a subclass) or construct it with a Runnable
  - Once you create a Thread instance, you can start it
  - Fairly low-level methods allow synchronization and inspection of threads

Asynchronous Programming

- Say we have a (fairly long-running) task that returns a result
  - E.g.: Authenticate using a remote service (e.g., Google, Facebook, GitHub, etc.)
- As a programmer, I want to:
  - Run the task in a background thread, while doing other things in the main thread
  - Do something with the result of the task, whenever it's ready
  - Handle any error that the task might throw
  - Specify a timeout, so that my program is not stuck waiting for the task, in case it runs indefinitely

Asynchronous Callbacks

- Instead of waiting for a function to return a result, tell it in advance what you want to do with its result
- That is:
  - When calling the function, pass a **callback** (the callback is just another function)
  - The function runs in a separate thread
  - Once the function is done, the callback is called (with function's result as its argument(s))

Synchronous vs. Asynchronous

Let's see the difference between synchronous and asynchronous code.
(*Note:* This example uses JavaScript)

```
try {

  data = syncFunc(someArg);
  // Do something with the data …

} catch (error){
  console.log(error);
}
```

```
asyncFunc(someArg,

  function(data){
    // Do something with the data …
  },

  function(error){
    console.log(error);
  }
);
```

Callbacks in JavaScript

- Functions might take a long time to complete
  - `var photo = downloadPhoto('http://coolcats.com/cat.gif')`
    `// photo is 'undefined'!`
- Don't want to block while the photo is downloading
  - Store the code that should run after the download is complete in a function (a callback)

```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto)

function handlePhoto (error, photo) {
  if (error) console.error('Download error!', error)
  else console.log('Download finished', photo)
}

console.log('Download started')
```

[http://callbackhell.com/]

Asynchronous Programming

More features/requirements:

- Chain tasks together
  - When a task is done, pass its result to the next task (and run the next task)
  - More complex “plumbing”, for example:
    - When a task is done, start three different tasks (in parallel)
    - Wait for multiple tasks to complete (all of them), and only then run the next task
    - Wait for the first of multiple tasks to complete, and pass its result to the next task
    - Pass exception down the chain, handle it and (possibly) continue to chain additional tasks
- Specify which threads to use for running background tasks
  - More control

Chaining Asynchronous Calls

```
asyncFunc1(someArg,
    function(result1){
        var x = doSomethingWith(result1);
        asyncFunc2(x,
            function(result2){
                doSomethingWith(result2);
            },
            function(error){
                console.log("f2 error: " + error);
            }
        );
    },
    function(error){
        console.log("f1 error: " + error);
    }
);
```

In the example on the left, we chain two asynchronous functions.

Essentially, we are programming the following logic (plus error handling):

Call `asyncFunc1` and, when its result is ready, call `asyncFunc2`.

Chaining Asynchronous Calls

As our chains get longer (and, possibly, more complex), the code becomes very hard to manage.

For example:

```
getData(function(a){
    getMoreData(a, function(b){
        getMoreData(b, function(c){
            getMoreData(c, function(d){
                getMoreData(d, function(e){
                    ...
                });
            });
        });
    });
});
```

Note: This pattern is known as *Callback Hell*

Drawbacks of multithreading

- Harder to trace and debug
- Exponential growth of complexity when multiple threads are running
- Higher probability of unexpected results (i.e., nondeterminism)
- Unintended dependencies (e.g., exception handling)

Promises (Futures)

- Asynchronous programming fits well with many of the tasks modern programmers deal with (because of the nature of the Internet)
- That being said, it doesn't always fit well with the syntax that modern programmers are used to
  - Chaining multiple methods can result in code that is very hard to read
  - Exceptions need to be handled more carefully
- The ***promise*** design pattern to the rescue...

Without promises

```
asyncFunc1(someArg,
  function(result1){
    var x = doSomethingWith(result1);
    asyncFunc2(x,
      function(result2){
        doSomethingWith(result2);
      },
      function(error){
        console.log("f2 error: " + error);
      }
    );
  },
  function(error){
    console.log("f1 error: " + error);
  }
);
```

vs.

With promises

```
asyncFunc1(someArg)
  .then(function(result1){
    return doSomethingWith(result1);
  })
  .then(function(result2){
    doSomethingWith(result2);
  })
  .catch(function(error){
    console.log("Caught " + error)
  });
```

Promise

- ***Promise*** (aka *future*, *delay* or *deferred*) is an object that represents a result that is initially unknown
- Different languages use different constructs and terminology (e.g., Java uses CompletableFuture), but the basic idea is the same:
    - Decouple a computation from the value it produces
    - Compose computations (i.e., functions) together (before you run/execute them)
    - A nice tutorial on using CompletableFuture (and another one)
    - Can promises fail to complete successfully? How do we handle that?
- Where can we leverage the asynchronous nature of promises/futures?
    - Computationally intensive processes (e.g., scientific calculations)
    - Manipulating large data structures
    - Remote method calls (downloading files, web services, etc.)

Java Threads - More Control

- Threads are resources (just like memory or CPU time)
- Sometimes we want more control over how resources are managed

  For example:
  - Parallelism
  - Reusing threads (instead of creating new ones)
  - Managing cache (to optimize context switching)
  - etc.

Resource Pool

- Object Pool is a common design pattern (aka Resource Pool)
  - As the name suggests, we create an object that is responsible for managing a pool of resources
    - The resource pool manages the ***lifecycle*** of resources (e.g., construction, destruction, etc.)
  - Two of the most common examples for resources that can benefit from pooling:
    - Threads
    - Database connections
  - ScheduledThreadPoolExecutor is an example of a thread pool that comes built-in with Java
    - Here is really nice tutorial you might find useful

Code Example

- This commit history of this short code example shows a few of the concepts discussed in this lecture.

Example - Context

- An application needs to process hundreds of thousands of records and update them
- Process is a multi-step transaction and the order of execution is important
- Records are grouped by “Accounts” that are independent of each other
- Records may need other resources that are shared among them
- Speed is not important, but success and correctness are critical

Example - Initial Solution

- Creating a new thread for every account (i.e., pool)
- Chaining multiple steps using promises (futures)
- Shared resources are locked when they are needed

Example - Problems

- Unpredictable results
  - Dependent on the order of execution of threads for each account
  - Deadlocks
- Complicated process to handle unexpected results
- High resource consumption

Example - Final Solution

- Simplified multiple steps into two
  - Minimized the number of chains
- Made every process completely independent
  - Architected around shared resources
- Grouped all DML (Data Manipulation Language) operations into one final “clean-up” process
  - Created one shared final cleanup (similar to a shutdown hook)

Summary

- We use multithreading to perform multiple tasks simultaneously
- We can use *thread-pools* to manage our threads.
  Common example:
  - The UI thread (a pool with a single thread)
  - Worker threads (use for long-running background tasks)
- When working with asynchronous code:
  - Use ***promises*** to chain functions together (i.e., create pipelines) and keep the code easily maintainable
  - When executing an asynchronous pipeline, use a thread-pool for long-running, background tasks (e.g., network request, or reading from disk)