Concurrency in Java

OOP week 10 lectures

Dr. **Madasar** Shah

December 2021

This session

- Writing threaded programs in Java
- Anti-patterns: race conditions, deadlock, starvation, live lock
- Patterns: Producer-Consumer, ThreadPool patterns and libraries

Writing Threaded Programs in Java

Golden rules:

1. write parallel code in a `.run()` method
2. do not call `.run()` directly, instead create a Thread and `.start()` it
3. Avoid concurrency anti-pattern: race conditions, deadlock, starvation etc
4. Use concurrency patterns and library code where possible

Two approaches to defining theaded code:

- extend Thread: class cannot extend any another class
- implement Runnable: more *idiomatic* Java approach

Implements runnable example: Concurrent Cashier System

```java
class Cashier implements Runnable {
  ...
  public void run () {
    ...
    for (int i = 0; i < 1000000; i++) {
        account.increment();
        account.decrement();
    }
    ...
  }
  ...
  public static void main(String args[]) {
  ...
      staff[i] = new Thread(new Cashier(args[i], budget));
      staff[i].start();
  ...
}
```

try it on codepad!

extend Thread vs implement Runnable differences

- only significant difference between implementing Runnable and extending Thread
- only ever `extend Thread` in you are specialising the thread's behaviour.
- in practice: done either ways, prefer implement Runnable

Question: what does `Thread.join()` do in the Concurrent Cashier System from the last lecture?

Concurrency: challenges, examples and definitions

- race conditions: Cashier System ???
- mutual exclusion problem: Cashier System ???
- critical sections: Cashier System ???
- deadlock
- live lock
- starvation

Concurrency: challenges, examples and definitions

- race conditions: interleaving of balance++
- mutual exclusion problem: modification of `int balance`
- critical sections: `Account.increment()` and `.decrement()`
- deadlock
- live lock
- starvation

Question: in the fixed concurrent cashier system what happens if we have multiple Account objects (and multiple cashiers)?

Concurrency anti pattern: deadlock

Case Study: Dining Philosophers

- Dining.java (main method, creates Philosophers and forks)
- Philosopher.java (zero or more Threads, each has two forks)
- Object-s for forks (two forks required to start 'work' in thread)

Shared Objects Structure: `Dining.java`

```java
while(true) {
    synchronized (firstFork) {
        synchronized (secondFork) {
            ...
            sleep(4); //some work.
            ...
        }
    }
}
```

try it on codepad!

Deadlock: definition

A deadlock may happen in a **where multiple independent threads** can access **shared resources**. Deadlock occurs when at **least two processes** are **waiting for the other to release a resource**. None of the processes can make any progress.

Deadlock requires four specific circumstances:

- A shared resource that cannot be used by more than one thread at a time
- A thread holding one resource may request another resource
- Resources cannot be released without an action of the thread
- One thread is waiting for a second thread to release a resource, whilst the second thread is waiting for the first thread to release a different resource

`https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-09766-4_282`

Deadlock: strategies to resolve deadlock

A deadlock may be solved by using one (or more) of the following strategies:

- **Avoid Unnecessary Locks**: We should use locks only for those members on which it is required. If possible, keep your code free form locks. For example, instead of using synchronized ArrayList use the ConcurrentLinkedQueue.
- **Avoid Nested Locks**: Another way to avoid deadlock is to avoid giving a lock to multiple threads if we have already provided a lock to one thread.
- **Using Thread.join() Method**: You can get a deadlock if two threads are waiting for each other to finish indefinitely using thread join() to interrupt the thread. Use the maximum time the thread should finish.
- **Use Lock Ordering**: Assign a numeric value to each lock. Before acquiring the lock with a higher numeric value, acquire the locks with a lower numeric value.
- **Lock Time-out**: We can also specify the time for a thread to acquire a lock. If a thread does not acquire a lock, the thread may release all acquired locks and wait before retrying to acquire a lock.

Concurrency anti-pattern: Live lock definition, strategies

- **Live lock** - similar to deadlock, less common
- Key difference: threads waiting for lock can continue but not do any useful work
- A third similar concept: **starvation**, where some threads are able to do useful work but other thread are *starved* of priority
- Harder to identify starvation - tasks still get done eventually
- Solving live lock and starvation harder: try same strategies as deadlock, sometimes simpler apply a concurrency design pattern

More advanced concurrency : Design patterns and APIs for concurrency

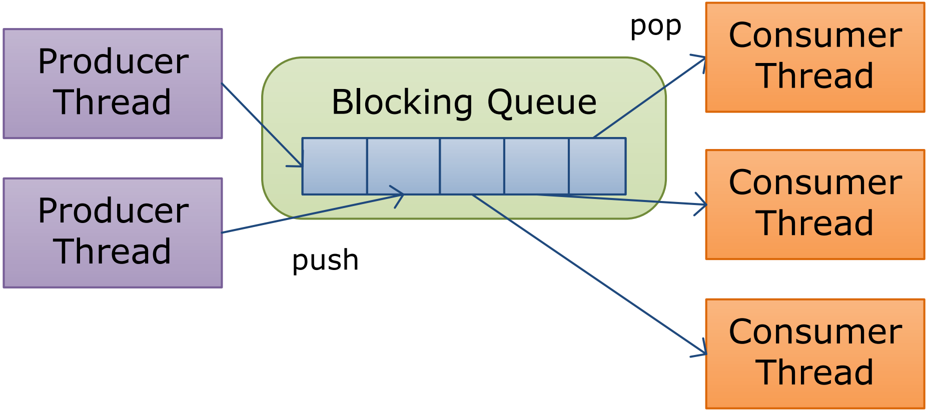
- Producer-Consumer: BlockingQueue / *workbox* example
- ThreadPool: ExecutorService / *LongTask* example

Thread pattern: producer consumer

Motivating Example iOS email client

- Server connection**s**
- Unpredictable network I/O
- Local mail database server
- Database, network and mail servers are independent
- How to store downloaded messages efficiently?

Structure



- Producer Thread(s)
- Consumer Thread(s)
- Blocking Queue(s)

Concurrency Pattern: Producer-Consumer / Blocking Queue

- Manager.java (implements Runnable, holds reference to BlockingQueue<Integer>, *puts* work)
- Worker.java (implements Runnable, holds reference to BlockingQueue<Integer>, *takes* work)
- main() creates BlockingQueue<Integer>, Managers and Workers

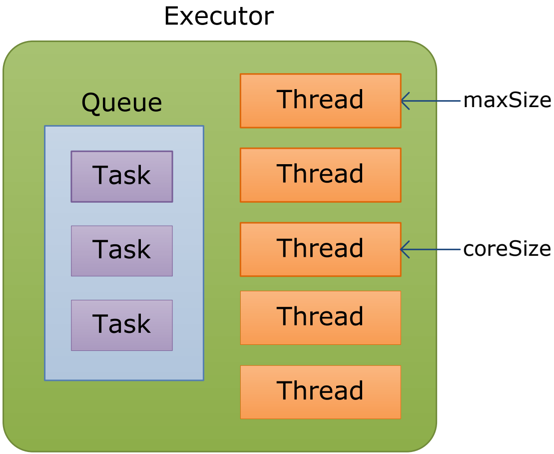
```
public class Main{
  public static void main(String args[]) {
    //work is done via a workbox
    BlockingQueue<Integer> workbox =
        new LinkedBlockingQueue<Integer>(1);
    // 1 = fixed size, artificially limited

    new Manager(workbox, 1);
    new Worker(workbox, 1);
    new Worker(workbox, 2);
  }
}                                        try it on codepad!
```

Concurrency Pattern: ThreadPool / ExecutorService

Motivating example: any kind of server

- Web, file, email, database, echo
- Listening for clients to connect
- Responding to multiple clients simultaneously
- With uniform requests/responses

Structure



- Executor(s)
- Pooled Threads
- Incoming Task Queue/List

Concurrency Pattern: ThreadPool / BlockingQueue class

- Main.java (creates ExecutorsService, LongTask)
- LongTask.java (independent threaded, tasks)

```java
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
...
  public static void main(String args[]) {
    ExecutorService tpe = Executors.newFixedThreadPool(3);

    tpe.submit(new LongTask(1));
    tpe.submit(new LongTask(2));
    tpe.submit(new LongTask(3));
    //can keep submitting to tpe

    tpe.shutdown();
  }
```

try it on codepad!

Thanks

- Questions?

With thanks to Martín Escardó.