Concurrency

Concurrent processing

Improve responsiveness

- Update result in small steps

Improve total processing time

Utilize CPU efficiently

Improve processing time

- Run I/O intensive tasks and CPU intensive tasks concurrently
- CPU is idling when waiting for I/O

Utilize multiple CPUs

- One thread is executed by one CPU
- Spread units of work over available CPUs
- Basic form of parallelism

Creating threads

- Thread creation and teardown is expensive
- Idling threads consume memory
- Competing threads hurt performance
- Maximum number of threads is limited

Executor framework

- API for concurrent task execution
- Support for thread pools
- Result bearing

Executor API

- Executor - describes a task
- Callable - describes a result bearing task
- Future - task's lifecycle representation
- Executors - thread pool factory methods

Creating executors

```
Executor exec = Executors.newFixedThreadPool(20);
```

```
Executor exec = Executors.newCachedThreadPool();
```

```
Executor exec = Executors.newSingleThreadExecutor();
```

Executing tasks

```
Executor exec = Executors.newFixedThreadPool(20);

Runnable task = new Runnable() {
    public void run() {
        //Do something
    }
};

exec.execute(task);
```

Callable

```
ExecutorService exec = Executors.newFixedThreadPool(10);
Callable<Integer> task = new Callable<Integer>() {
    public Integer call() throws Exception {
        //Calculate something
        return 5;
    }
};

Future<Integer> future = exec.submit(task);

//Do other tasks

Integer result = future.get();
```

CompletionService

- Submit tasks to the service
- Completed tasks are added to a blocking queue
- Handle each result when available

Creating a CompletionService

```
Executor exec = Executors.newFixedThreadPool(20);
CompletionService<Integer> completionService =
        new ExecutorCompletionService<Integer>(exec);

for(int i = 1; i < 100; i++) {
    final int number = i;

    Callable<Integer> task = new Callable<Integer>() {
        public Integer call() throws Exception {
            float sleep = Math.round(Math.random() * 10);
            TimeUnit.SECONDS.sleep((int)sleep);
            return number * number;
        }
    };

    completionService.submit(task);
}
```

Handling results

```
for(int i = 1; i < 100; i++) {
    Future<Integer> f = completionService.take();
    Integer result = f.get();
    System.out.println("Result: " + result);
}
```

- take() blocks until result is available
- take() returns a result as Future

Client Locking

Broken!

```
public class ListUtils<E> {
    public final List<E> list =
            Collections.synchronizedList(new ArrayList());

    public synchronized boolean putUnique(E x) {
        boolean unique = !list.contains(x);

        if (unique) {
            list.add(x);
        }

        return unique;
    }
}
```

Is this code thread safe?

Synchronizes on wrong lock

Corrected example

```
public boolean putUnique(E x) {
    synchronized (list) {
        boolean unique = !list.contains(x);

        if (unique) {
            list.add(x);
        }

        return unique;
    }
}
```

Concurrent Collections

- Normal collections are **not** thread safe
- But there are special thread safe implementations

ConcurrentHashMap

Weakly consistent

CopyOnWriteArrayList

Safe iteration

ThreadLocal

- Bind data to the current thread
- Ensures thread confinement

Data is not shared between threads

- Used heavily by frameworks

EJB Transactions

Thread safe version of otherwise not thread safe class

```
public class IdentityManager {
    private static ThreadLocal<User> user = new ThreadLocal<User>();

    public static boolean isLoggedIn() {
       return user.get() != null;
    }

    public static User getLoggedInUser() {
        if(user.get() == null) {
            return null;
        }
        return user.get();
    }

    public static void logIn(String username, String password) {
        user.set(new User(username));
    }

}
```

ReadWriteLock

synchronize doesn't make a difference
between reading and writing data

Prefer a ReadWriteLock when:

- ☐ Many reads, few modifications
- ☐ Reads take fairly long

ReadWriteLock

```
private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

Read locks: many concurrent read, but blocks writes

```
lock.readLock().lock();

lock.readLock().unlock();
```

Write locks: many concurrent read, but blocks writes

```
lock.writeLock().lock();

lock.writeLock().unlock();
```

CountDownLatch

- Block until a set of operations is completed
- Blocks on await()
- Unblocks when count == 0
- Decrease count with countDown()

CountDownLatch

create a CountDownLatch

```
final CountDownLatch countDownLatch = new CountDownLatch(NR_OF_RUNS);
```

decrease count

```
countDownLatch.countDown();
```

block until count reaches zero

```
countDownLatch.await();
```

Semaphore

- Initialized with *N* number of permits
  - Two methods

proceed() -> blocks until permit is available

release() -> releases a permit

Canceling tasks

The most simple cancelation uses a canceled flag

Flag must be volatile

Doesn’t work for blocking calls

Doesn’t work for very long running tasks

Cancel flag example

```
public class SimpleCancelFlag implements Runnable{
    private volatile boolean canceled;

    public void run() {
        while(!canceled) {
            //Do something non-blocking
        }
    }

    public void cancel() {
        canceled = true;
    }
}
```

Cancel flag & blocking calls

Broken!

```
public class SimpleCancelFlag implements Runnable {
    private volatile boolean canceled;
    private final BlockingQueue<Integer> queue =
            new ArrayBlockingQueue<Integer>(10);

    public void run() {
        while (!canceled) {
            try {
                queue.put((int) Math.random() * 10);
            } catch (InterruptedException e) {
            }
        }
    }

    public void cancel() {
        canceled = true;
    }
}
```

May block forever

Interruption

- Each thread has an interrupted status
- Set interrupted by calling interrupt()
  - Interrupt() does NOT interrupt the thread

But sets the interrupted flag

Tasks should have an interruption policy

Important blocking calls are interruption

Fixed cancelation example

```
public class SimpleCancelFlag extends Thread {
    private final BlockingQueue<Integer> queue =
            new ArrayBlockingQueue<Integer>(10);

    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            try {
                queue.put((int) Math.random() * 10);
            } catch (InterruptedException e) {
            }
        }
    }

    public void cancel() {
        interrupt();
    }
}
```

Implementing interruption

- If the task owns the thread
  - Cleanup
  - Swallow InterruptedException so the thread can exit

Implementing interruption

If the task doesn't own the thread;

either:

Propagate the exception

Restore interruption status

Sizing thread pools

- How many processors are available?
- What other resources are used by tasks?

Are tasks CPU intensive?

Are tasks waiting for I/O?

- Do tasks use a database connection pool?

Sizing thread pools

CPU intensive tasks

- size = number of CPUs + 1

```
Runtime.getRuntime().availableProcessors();
```

Sizing thread pools

I/O intensive tasks

- ☐ Profile to see what threads are doing
- ☐ Increase pool size when threads are waiting
- ☐ Decrease pool size when threads are competing for resources

Sizing thread pools

Connection pools

- When using a connection pool
- Thread pool size is limited by connection pool size