Thread Control and Deadlocks

1. Thread Interruption

Thread interruption is a mechanism in Java that allows one thread to interrupt the execution of another thread. It's a cooperative process, meaning that the interrupted thread must be designed to respond to interruption.

Key Points:

- Interruption is requested by calling the `interrupt()` method on a Thread object.
- The interrupted thread can check its interrupted status using `Thread.interrupted()` or `isInterrupted()`.
- Many blocking methods (like `sleep()`, `wait()`, `join()`) throw `InterruptedException` when interrupted.

Example:

```java
public class InterruptExample implements Runnable {
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println("Working...");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new InterruptExample());
        thread.start();
        Thread.sleep(5000);
        thread.interrupt();
    }
}
```

2. Fork/Join Framework

The Fork/Join framework, introduced in Java 7, is designed for parallel execution of recursive, divide-and-conquer algorithms.

Key Components:

1. `ForkJoinPool`: An ExecutorService for running ForkJoinTasks.
2. `RecursiveTask<V>`: A task that returns a result.

3. `RecursiveAction`: A task that doesn't return a result.

How it works:

1. The problem is divided into smaller subtasks.
2. Subtasks are solved recursively.
3. Results of subtasks are combined to produce the final result.

Example: Parallel Sum Calculation

```java
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class ParallelSum extends RecursiveTask<Long> {
    private static final int THRESHOLD = 10000;
    private long[] array;
    private int start;
    private int end;

    public ParallelSum(long[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Long compute() {
        if (end - start <= THRESHOLD) {
            long sum = 0;
            for (int i = start; i < end; i++) {
                sum += array[i];
            }
            return sum;
        } else {
            int middle = (start + end) / 2;
            ParallelSum left = new ParallelSum(array, start, middle);
            ParallelSum right = new ParallelSum(array, middle, end);
            left.fork();
            long rightResult = right.compute();
            long leftResult = left.join();
            return leftResult + rightResult;
        }
    }

    public static void main(String[] args) {
        long[] array = new long[100000];
        for (int i = 0; i < array.length; i++) {
            array[i] = i;
        }
        ForkJoinPool pool = new ForkJoinPool();
        ParallelSum task = new ParallelSum(array, 0, array.length);
        long sum = pool.invoke(task);
```

```
        System.out.println("Sum: " + sum);
    }
}
```

3. Deadlock Prevention Techniques

Deadlock occurs when two or more threads are unable to proceed because each is waiting for the other to release a resource. Here are some techniques to prevent deadlocks:

1. Lock Ordering

Ensure that all threads acquire locks in the same order. This prevents circular wait conditions.

```java
public void method() {
    synchronized(lock1) {
        synchronized(lock2) {
            // Critical section
        }
    }
}
```

2. Lock Timeout

Use `tryLock()` with a timeout to avoid indefinite waiting.

```java
if (lock.tryLock(1, TimeUnit.SECONDS)) {
    try {
        // Critical section
    } finally {
        lock.unlock();
    }
} else {
    // Handle lock acquisition failure
}
```

3. Deadlock Detection

Implement a deadlock detection algorithm and release locks if a potential deadlock is detected.

4. Resource Allocation Graph

Use a resource allocation graph to analyze and prevent deadlocks in complex systems.

5. Avoid Nested Locks

Minimize the use of nested locks. If necessary, use techniques like lock ordering.

6. Use `java.util.concurrent`

Utilize higher-level concurrency utilities like `ReentrantLock`, `Semaphore`, and `CountDownLatch` which provide more robust locking mechanisms.

7. Release Locks in Finally Blocks

Always release locks in finally blocks to ensure they are released even if an exception occurs.

```java
Lock lock = new ReentrantLock();
try {
    lock.lock();
    // Critical section
} finally {
    lock.unlock();
}
```

By understanding and implementing these concepts and techniques, you can write more efficient, robust, and deadlock-free multithreaded Java applications.