# Parallel Programming

[Hassan Almosa]

June 2025

# Contents

# 1 Build and Run Instructions

`cd` into project's root directory and compile the source files. Make sure the 'jargs.gnu.CmdLineParser' is included in the classpath. The compiled classes will be placed in the 'out' directory. Ignore any deprecation warnings.

Compilation command:

```
#Checkpoints 4.1 and 4.2
javac -d out src/cp3/lab04/expandablearray/*.java

#Checkpoints 4.3 and 4.4
javac -d out src/cp3/lab04/crypt/JCrypt.java src/cp3/lab04/crypt/JCryptUtil.java
    src/jargs/gnu/CmdLineParser.java
```

Listing 1: Compilation Command

To run with different threading implementations, follow instructions in the code comments by commenting/uncommenting relevant sections, recompiling, and running again.

Example run commands:

```
#Checkpoints 4.1 and 4.2
java -cp out cp3.lab04.expandablearray.ExpandableArrayDriver

#Checkpoints 4.3 and 4.4
# Single-thread per file and serial one-thread execution
java -cp out cp3.lab04.crypt.JCrypt -d enigma -s prac4-secrets/*.jpg.encrypted

# Multi-threaded execution (e.g., 4 threads)
java -cp out cp3.lab04.crypt.JCrypt -d enigma -t 4 prac4-secrets/*.jpg.encrypted
```

Listing 2: Example Run Commands

# 2 Checkpoint 4.1: ExpandableArray Concurrency Issues

`ArrayManip` class that extends `Thread` was created and operates on a shared `ExpandableArray` object. By not implementing `synchronized` keyword from all methods, multiple threads simultaneously access the same array instance without synchronisation, this creates race conditions because operations like array expansion and size increment (`size++`) are not atomic, resulting induced crashes: `ArrayIndexOutOfBoundsException` and `NullPointerException` due to inconsistency and unsafe shared memory access.

```
 public ArrayManip(ExpandableArray array) {
        this.array = array;

    }
    public void run(){
        for (int i = 0; i < 100; i++){
            try {
                array.add(i);

                // Add null pointer exception condition - try to remove when array
    might be empty
                if (i % 10 == 0 && array.size() > 0) {
                    array.removeLast();
                }
            }
```

```
            catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Thread " + Thread.currentThread().getId() +
                                   ": ArrayIndexOutOfBoundsException at iteration " +
    i + " - " + e.getMessage());
            }
            catch (NullPointerException e) {
                System.out.println("Thread " + Thread.currentThread().getId() +
                                   ": NullPointerException at iteration " + i + " - "
    + e.getMessage());
            }
}
```

Listing 3: ArrayManip Thread Implementation

**Sample Run Output:**

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 116 out
    of bounds for length 115
        at cp3.lab04.expandablearray.ExpandableArray.add(ExpandableArray.java:70)
        at cp3.lab04.expandablearray.ExpandableArrayDriver.main(
    ExpandableArrayDriver.java:34)
```

Listing 4: Concurrency Issues Demonstration 1

To simulate another type of error

```
public class ArrayManip extends Thread {

    ExpandableArray array;

    public ArrayManip(ExpandableArray array) {
        this.array = array;

    }
    public void run(){
        for (int i = 0; i < 200; i++){
            try {
                array.add(i);
                // Add race conditions - random size events
                if (i % 10 == 0 && array.size() > 5) {
                    array.get(array.size() - 1); // Read last element
                }
                //remove random
                if (i % 15 == 0 && array.size() > 10) {
                    array.removeLast(); // Remove last element
                }
            }
            catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Thread " + Thread.currentThread().getId() +
                                   ": ArrayIndexOutOfBoundsException at iteration " +
    i + " - " + e.getMessage());
            }
            catch (NullPointerException e) {
                System.out.println("Thread " + Thread.currentThread().getId() +
                                   ": NullPointerException at iteration " + i + " - "
    + e.getMessage());
            }
            catch (Exception e) {
                System.out.println("Thread " + Thread.currentThread().getId() +
                                   ": Unexpected exception at iteration " + i + " - "
```

```
     + e.getMessage());
33             }
34         }
35     }
```

Listing 5: ArrayManip Thread Implementation

**Sample Run Output:**

```
1  java -cp src cp3.lab04.expandablearray.ExpandableArrayDriver
2  Starting am …
3  Starting am2 …
4  Starting am3 …
5  Starting am4 …
6  Starting am5 …
7  Starting am6 …
8  Starting am7 …
9  Starting am8 …
10 Starting am9 …
11 Starting am10 …
12 Thread 18 completed.
13 Thread 23 completed.
14 Thread 16 completed.
15 Size now: 1
16 Thread 19 completed.
17 Main thread adding 1000 elements …
18 final size: 1000
19 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 Thread 17: NullPointerException
       at iteration 0 - Expandable Array Empty
20 Thread 22: NullPointerException at iteration 1 - Expandable Array Empty
```

Listing 6: Concurrency Issues Demonstration 2

# 3 Checkpoint 4.2: ExpandableArray with Synchronisation

## 3.1 Thread Policy

The synchronised version uses the `synchronized` Java implementation keyword on all `ExpandableArray` methods, and launches the multiple threads forked from main by calling `start()`, this limits only one thread to access the shared array at a time. This prevents race conditions while still allowing concurrent execution, enabling somewhat of a thread safety.

**Sample Run Output:**

```
1  Thread 18 completed.
2  Thread 15 completed.
3  Thread 21 completed.
4  Thread 19 completed.
5  Thread 17 completed.
6  Size now: 1354
7  Main thread adding 1000 elements …
8  final size: 2620
9   …  … 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19 21 22 23 24 25 26 27 28 29 31 32
       33 34 35 36 37 38 39 41 42 43 44 45 46 47 48 49 51 52 53 54 55 56 57 58 59 61
      62 63 64 65 66 67 68 69 71 72 73 74 75 76 77 78 79 81 82 83 84 85 86 87 88 89 91
       92 93 94 95 96 97 98 99 101 102 103 104 105 106 107 108 109 111 112 113 114 115
       116 117 118 119 121 122 123 124 125 126 127 128 129 131 132 133 134 135 136 137
       138 139 141 142 143 144 145 146 147 148 149 151 152 153 154 155 156 157 158 159
       161 162 163 164 165 166 167 168 169 171 172 173 174 175 176 177 178 179 181 182
```

```
 183 184 185 186 187 188 189 191 192 193 194 195 196 197 198 199 1 2 3 4 5 6 7 8
 9 11 12 13 14 15 16 17 18 19 21 22 23 24 25 26 27 28 29 31 32 33 34 35 36 37 38
 39 41 42 43 44 45 46 47 48 49 51 52 53 54 55 56 57 58 59 61 62 63 64 65 66 67
68 69 71 72 73 74 75 76 77 78 79 81 82 83 84 85 86 87 88 89 91 92 93 94 95 96 97
 98 99 101 102 103 104 105 106 107 108 109 111 112 113 114 115 116 117 118 119
121 122 123 124 125 126 127 128 129 131 132 133 134 135 136 137 138 139 141 142
143 144 145 146 147 148 149 151 152 153 154 155 156 157 158 159 161 162 163 164
165 166 167 168 169 171 172 173 174 175 176 177 178 179 181 182 183 184 185 186
187 188 189 191 192 193 194 195 196 197 198 199 1 2 3 4 5 6 7 8 9 11 12 13 14 15
 16 17 18 19 21 22 23 24 25 26 27 28 29 31 32 33 34 35 36 37 38 39 41 42 43 44
45 46 47 48 49 51 52 53 54 55 56 57 58 59 61 62 63 64 65 66 67 68 69 71 72 73 74
 75 76 77 78 79 81 82 83 84 85 86 87 88 89 91 92 93 94 95 96 97 98 99 101 102
103 104 105 106 107 108 109 111 112 113 114 115 116 117 118 119 121 122 123 124
125 126 127 128 129 131 132 133 134 135 136 137 138 139 141 142 143 144 145 146
147 148 149 151 152 153 154 155 156 157 158 159 161 162 163 164 165 166 167 168
169 1 171 2 172 3 173 4 174 5 175 6 176 7 177 8 178 9 179 10 12 181 13 182 14
183 15 184 16 185 17 186 18 187 19 188 189 21 190 22 23 24 26 191 27 192 28 193
29 194 30 196 31 197 32 198 33 199 34 35 36 37 38 39 41 42 43 44 45 46 47 48 49
51 52 53 54 55 56 57 58 59 61 62 63 64 65 66 67 68 69 71 72 73 74 75 76 77 78 79
 81 82 … ..
```

Listing 7: Synchronized ExpandableArray Output 1(Cropped)

```
… . 23012 23013 23014 23015 23016 23017 23018 23019 23020 23021 23022 23023 23024
23025 23026 23027 23028 23029 23030 23031 23032 23033 23034 23035 23036 23037
23038 23039 23040 23041 23042 23043 23044 23045 23046 23047 23048 23049 23050
23051 23052 23053 23054 23055 23056 23057 23058 23059 23060 23061 0 23062 1
23063 2 23064 3 4 23065 5 23066 6 23067 7 23068 8 23069 9 23070 10 23071 11
23072 12 23073 13 23074 14 23075 15 23076 16 23077 17 23078 18 23079 19 23080 20
 23081 21 23082 22 23083 23 23084 24 23085 25 23086 26 23087 27 23088 28 23089
29 23090 30 23091 31 23092 32 23093 33 23094 34 23095 35 23096 36 23097 37 23098
 38 23099 39 23100 40 23101 41 23102 42 23103 43 23104 44 23105 45 23106 46
23107 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 23108 62 23109 63 23110 64
23111 65 23112 66 23113 67 23114 68 23115 69 23116 70 23117 71 23118 72 23119 73
 23120 74 23121 75 23122 76 23123 77 23124 78 79 80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99   … ..
```

Listing 8: Synchronized ExpandableArray Output 2(Cropped)

# 4 Checkpoint 4.3: One Thread Per File

This checkpoint implements concurrent file processing where each file is handled by a dedicated thread. Multiple synchronisation methods were attempted and measure total execution time was measured. Performance was measured by running each set of 8 images decryption 3 times for all tests across *checkpoints 4.3* and *4.4*

## 4.1 Thread Policy

Each thread operates independently on its assigned file, the concurrent worker threads are forked from main thread and awaits for their completion to join main thread. This approach may be not optimal when number of files > available CPU cores.

## 4.2 Baseline - Serial Decryption

Baseline performance of decrypting supplied 8 images when processing files serially, one after another in a single thread.

```java
    static class SingleFileThread extends Thread {
        private JCryptUtil.Options opts;
        private int index;

        public SingleFileThread(JCryptUtil.Options opts, int index) {
            this.opts = opts;
            this.index = index;
        }

        @Override
        public void run() {
            try {
                process(opts, i);
            } catch (JCryptUtil.Problem e) {
                System.err.println("ERROR in thread: " + e.getMessage());
            }
        }
    }
```

Listing 9: `SingleFileThread` class

The `` `main` `` method iterates through the input files and calls the `` `process` `` method for each file. Time measurement is done for the entire loop.

```java
    public static void main(String[] args) {
        JCryptUtil.Options opts = JCryptUtil.parseOptions(args);
        long starttime = System.nanoTime();

        List<Thread> threads = new ArrayList<>();
        for (int i = 0; i < opts.filenames.length; i++) {
            Thread t = new SingleFileThread(opts, i);
            threads.add(t);
            t.start();}
        // Wait for all threads to complete
        for (Thread t : threads) {
            t.join();}
        System.out.println("Time taken (with join): " +
            (System.nanoTime()-starttime)/1000000000.0 + "s"); }
```

Listing 10: `main()` Method with `join()`

Total time of multiple runs of the serial decryption for all files, preformed 3 different times.

```
java -cp out cp3.lab04.crypt.JCrypt -d enigma -s prac4-secrets/*.jpg.encrypted
Decrypting prac4-secrets/classified.jpg.encrypted
Decrypting prac4-secrets/confidential.jpg.encrypted
Decrypting prac4-secrets/dangerous.jpg.encrypted
Decrypting prac4-secrets/hushhush.jpg.encrypted
Decrypting prac4-secrets/illegal.jpg.encrypted
Decrypting prac4-secrets/private.jpg.encrypted
Decrypting prac4-secrets/restricted.jpg.encrypted
Decrypting prac4-secrets/topsecret.jpg.encrypted
Time taken (Serial): 5.426980792s
Time taken (Serial): 9.224711125s
Time taken (Serial): 12.579783416s
```

Listing 11: Serial Decryption Test Runs

## 4.3 Concurrent Processing with One Thread Per File

Three synchronisation mechanisms were implemented to coordinate thread completion: `Thread.join()`, `CountDownLatch`, and `CyclicBarrier`.

### 4.3.1 `Thread.join()` Implementation

Uses a basic `SingleFileThread` class where the main thread waits for each worker thread using 'join()'.

```
1
2       public static void main(String[] args) {
3
4           JCryptUtil.Options opts = JCryptUtil.parseOptions(args);
5           long starttime = System.nanoTime();
6
7           try {
8               for (int i = 0; i < opts.filenames.length; i++) {
9                   process(opts, i);
10                  System.out.println("Time taken (Serial Decryption): " + (System.
    nanoTime()-starttime)/1000000000.0 + "s");
11
12              }
13          } catch (JCryptUtil.Problem e) {
14              System.err.println("ERROR: " + e.getMessage());
15              System.exit(2);}
16                  else {
17                  //threads list to hold all threads
18                  List<Thread> threads = new ArrayList<>();
19
20                  for (int i = 0; i < opts.filenames.length; i++) {
21
22                      // Option A: enable/disable for Regular using join()
23                      Thread t = new SingleFileThread(opts, i);
24
25                      threads.add(t);
26                      t.start();
27                  }
28                   Option A: enable/disable t.join() (default)
29                  for (Thread t : threads) {
30                    t.join(); // Wait for all threads to complete
```

Listing 12: `main` method

Capturing the total execution time through a print statement after threads call loop was challenging, as main would execute before the statement. Thread synchronisation using `join()` was necessary as the main thread must wait for all worker threads to complete. Time was captured manually using a stop watch and yielded similar results. Debugging this issue was interesting as it was a concurrency issue.

**Sample Runs Output:**

```
1 java -cp out cp3.lab04.crypt.JCrypt -d enigma prac4-secrets/*.jpg.encrypted
2 Time taken (with join): 4.450259292s
3 Time taken (with join): 12.293305542s
```

Listing 13: One Thread Per File with 'join()' Test Runs

### 4.3.2 `CountDownLatch` Implementation

A `` `CountDownLatch` `` was used for alternate synchronisation options. The latch is initialised with the number of files, each thread calls `` `countDown()` `` upon completion, and the main thread calls `` `await()` `` to block until all threads finish.

```
    static class SingleFileThreadWithLatch extends Thread {
        private JCryptUtil.Options opts;
        private int index;
        private CountDownLatch latch;

        public SingleFileThreadWithLatch(JCryptUtil.Options opts, int index,
    CountDownLatch latch) {
            this.opts = opts;
            this.index = index;
            this.latch = latch;
        }

        @Override
        public void run() {
            try {
                process(opts, index);
            } catch (JCryptUtil.Problem e) {
                System.err.println("ERROR in thread: " + e.getMessage());
            } finally {
                latch.countDown();
            }
        }
    }
```

Listing 14: CountDownLatch Thread Implementation

```

    public static void main(String[] args) {
            …

                else {
                //threads list to hold all threads
                List<Thread> threads = new ArrayList<>();


                // Option B: enable/disable CountDownLatch (uncomment to use
    instead of join() default)
                 CountDownLatch latch = new CountDownLatch(opts.filenames.length);

                for (int i = 0; i < opts.filenames.length; i++) {

                    // Option B: enable/disable for CountDownLatch
                    Thread t = new SingleFileThreadWithLatch(opts, i, latch);

                    threads.add(t);
                    t.start();
                }
                 Option B: enable/disable CountDownLatch
                 latch.await(); // await for all threads to finish
                 System.out.println("Time taken (with CountDownLatch): " + (System.
    nanoTime()-starttime)/1000000000.0 + "s");
                 }
```

Listing 15: `main` method latch implementation

**Sample Runs Output:**

```
 java -cp out cp3.lab04.crypt.JCrypt -d enigma prac4-secrets/*.jpg.encrypted
..    ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken (with CountDownLatch): 10.090022334s
Time taken (with CountDownLatch): 16.085607541s
```

Listing 16: One Thread Per File with 'CountDownLatch' Test Runs

### 4.3.3 Implementation using `CyclicBarrier`

A `CyclicBarrier` was also tested although not ideal for our use case. The barrier is initialised with the number of worker threads plus one (for the main thread). Each worker thread calls `barrier.await()` after processing its file, and the main thread also calls `barrier.await()`. All threads will wait until all threads has arrived.

```

 static class SingleFileThreadWithBarrier extends Thread {
        private JCryptUtil.Options opts;
        private int index;
        private CyclicBarrier barrier;

        public SingleFileThreadWithBarrier(JCryptUtil.Options opts, int index,
    CyclicBarrier barrier) {
            this.opts = opts;
            this.index = index;
            this.barrier = barrier;
        }

        @Override
        public void run() {
            try {
                process(opts, index);
                barrier.await(); // Wait for all threads to complete
            } catch (JCryptUtil.Problem e) {
                System.err.println("ERROR in thread: " + e.getMessage());
            } catch (Exception e) {
                System.err.println("Barrier error: " + e.getMessage());
            }
        }
    }
```

Listing 17: `SingleFileThreadWithBarrier.java` Class

```

    public static void main(String[] args) {

                else {
                //threads list to hold all threads
                List<Thread> threads = new ArrayList<>();

                // Option C: enable/disable CyclicBarrier
                 CyclicBarrier barrier = new CyclicBarrier(opts.filenames.length +
    1);

                for (int i = 0; i < opts.filenames.length; i++) {

```

```
                    // Option C: enable/disable For CyclicBarrier
                    Thread t = new SingleFileThreadWithBarrier(opts, i, barrier);

                    threads.add(t);
                    t.start();
                }


                // Option C: CyclicBarrier enable/disable
                barrier.await();
                System.out.println("Time taken (with CyclicBarrier): " + (System.
    nanoTime()-starttime)/1000000000.0 + "s");
        …
```

Listing 18: `main` method barrier implemntation

**Sample Run Output:**

```
 java -cp out cp3.lab04.crypt.JCrypt -d enigma prac4-secrets/*.jpg.encrypted
..    ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken (with CyclicBarrier): 11.947831333s
Time taken (with CyclicBarrier): 11.696912792s
Time taken (with CyclicBarrier): 12.750577958s
```

Listing 19: One Thread Per File with ‘CyclicBarrier‘ Test Runs

# 5 Checkpoint 4.4: Specified Number of Threads

This checkpoint implements thread safety approaches and thread pooling approach where a fixed number of worker threads process files from a shared queue, and with multiple synchronisation methods as attempted before. Performance is heavily impacted by selecting -s option to write files, and by current OS process.

Testing was performed on an *ARM64 M1 MacBook* with 8 cores. Time measurements showed significant variability due to background processes and the concurrent nature of the operations, making precise performance comparison challenging and would require more containerised identical enviornment and ideal benchmarking tools for accurate preformance measurment. However, the test runs of the various implemntations and synchronisation methods yielded somewhat comparable results, and noticeable hinderence of machine preformance (and occasional full OS crashes) upon high increase of number of threads. Also, its important to note the decryption/encryption password length was significant factor of performance.

## 5.1 Thread Policy

The threading policy for this checkpoint uses a fixed number of worker threads (specified by the `-t` option) that compete for files from a shared work queue. This approach provides better resource management and prevents thread explosion when processing many files. Thread safety was utilised by using `AtomicInteger` and a counter, and also `ReenterantLock` approach was tried, and various synchronisation thread methods were tried.

## 5.2 Baseline Implementation

The baseline implementation uses `AtomicInteger` for thread-safe file indexing and `Thread.join()` for coordination.

```
1
2     static class MultiThread extends Thread {
3         private JCryptUtil.Options opts;
4         private AtomicInteger fileCounter;
5         private int totalFiles;
6
7         public MultiThread(JCryptUtil.Options opts, AtomicInteger fileCounter, int
    totalFiles) {
8             this.opts = opts;
9             this.fileCounter = fileCounter;
10            this.totalFiles = totalFiles;
11        }
12
13        @Override
14        public void run() {
15            int currentFileIndex;
16            while ((currentFileIndex = fileCounter.getAndIncrement()) < totalFiles)
     {
17                try {
18                    process(opts, currentFileIndex);
19                } catch (JCryptUtil.Problem e) {
20                    System.err.println("ERROR in thread: " + e.getMessage());
21                }
22            }
23        }
24    }
```

Listing 20: Baseline Thread Pool Implementation

```
1
2     public static void main(String[] args) {
3             …
4              try {
5             if (opts.threads > 0) {
6              …
7                         for (int i = 0; i < numberOfThreads; i++) {
8
9                 // Option A: enable/disable for MultiThread using join()
10                    Thread multiThread = new MultiThread(opts, fileCounter, opts.
    filenames.length);
11
12                                for (Thread multiThread : multiThreads) {
13                    multiThread.join(); // This is also how we wait for
    ReentrantLock threads too, as they exit run() and join main
14                 }
15                System.out.println("Time taken: " + (System.nanoTime()-starttime)
    /1000000000.0 + "s");
```

Listing 21: `main` method base implementation

**Sample Runs Output:**

Using 4 threads

```
 java -cp out cp3.lab04.crypt.JCrypt -d enigma -t 4 prac4-secrets/*.jpg.encrypted
..   ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken: 7.737602042
Using 4 threads for 8 files
Time taken: 12.10110025s
Using 4 threads for 8 files
Time taken: 10.052850916s
Using 4 threads for 8 files
```

Listing 22: Updated test script commands

Using 8 threads

```
 java -cp out cp3.lab04.crypt.JCrypt -d enigma -t 8 prac4-secrets/*.jpg.encrypted
..   ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken: 8.886693916s
Using 8 threads for 8 files
Time taken: 9.019521s
Using 8 threads for 8 files
Time taken: 7.649646125s
Using 8 threads for 8 files
```

Listing 23: Updated test script commands

## 5.3 CountDownLatch Implementation

The `CountDownLatch` approach uses a countdown mechanism where each thread decrements the latch upon completion. The main thread waits until all worker threads signal completion.

```
    // For CountDownLatch with multiple threads
    static class MultiThreadWithLatch extends Thread {
        private JCryptUtil.Options opts;
        private AtomicInteger fileCounter;
        private int totalFiles;
        private CountDownLatch latch;

        public MultiThreadWithLatch(JCryptUtil.Options opts, AtomicInteger
    fileCounter, int totalFiles, CountDownLatch latch) {
            this.opts = opts;
            this.fileCounter = fileCounter;
            this.totalFiles = totalFiles;
            this.latch = latch;
        }
        @Override
        public void run() {
            try {
                int currentFileIndex;
                while ((currentFileIndex = fileCounter.getAndIncrement()) <
    totalFiles) {
                    try {
                        process(opts, currentFileIndex);
                    } catch (JCryptUtil.Problem e) {
                        System.err.println("ERROR in thread: " + e.getMessage());
                    }
```

```
                }
            } finally {
                latch.countDown(); // Signal completion
            }
        }
```

Listing 24: CountDownLatch Thread Implementation

```
    public static void main(String[] args) {
            …
             try {
            if (opts.threads > 0) {
            ..
                      // Option B: CountDownLatch (uncomment to use instead of join())
                CountDownLatch latch = new CountDownLatch(numberOfThreads); // numberOfThreads would be the count of threads actually created
                                  // Option B: enable/disable for CountDownLatch (uncomment to use instead of join() default)
                   Thread multiThread = new MultiThreadWithLatch(opts, fileCounter, opts.filenames.length, latch);
                                    // Option B: CountDownLatch
                latch.await();
               System.out.println("Time taken (latch): " + (System.nanoTime()-starttime)/1000000000.0 + "s");
```

Listing 25: `main` method latch implementation

**Sample Runs Output:**

Using 4 threads

```
-cp out cp3.lab04.crypt.JCrypt -d enigma -t 4 prac4-secrets/*.jpg.encrypted
..   ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken (barrier):  7.772499209s
Using 4 threads for 8 files
Time taken (latch): 8.469825834s
Using 4 threads for 8 files
Time taken (latch): 8.8708475s
Using 4 threads for 8 files
```

Listing 26: Updated test script commands

Using 8 threads

```
java -cp out cp3.lab04.crypt.JCrypt -d enigma -t 8 prac4-secrets/*.jpg.encrypted
..   ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken (latch): 7.710039417s
Using 8 threads for 8 files
Time taken (latch): 7.80643075s
Using 8 threads for 8 files
Time taken (latch): 7.737509s
Using 8 threads for 8 files
```

Listing 27: Updated test script commands

### 5.4 CyclicBarrier Implementation

The `CyclicBarrier` approach coordinates threads by requiring all threads to reach a common barrier point before any can proceed. Each thread calls `barrier.await()` upon completion.

```
static class MultiThreadWithBarrier extends Thread {
    private JCryptUtil.Options opts;
    private AtomicInteger fileCounter;
    private int totalFiles;
    private CyclicBarrier barrier;

    public MultiThreadWithBarrier(JCryptUtil.Options opts, AtomicInteger
    fileCounter, int totalFiles, CyclicBarrier barrier) {
        this.opts = opts;
        this.fileCounter = fileCounter;
        this.totalFiles = totalFiles;
        this.barrier = barrier;
    }

    @Override
    public void run() {
        try {
            int currentFileIndex;
            while ((currentFileIndex = fileCounter.getAndIncrement()) < totalFiles)
     {
                try {
                    process(opts, currentFileIndex);
                } catch (JCryptUtil.Problem e) {
                    System.err.println("ERROR in thread: " + e.getMessage());
                }
            }
            barrier.await(); // Wait for all threads to complete
        } catch (Exception e) {
            System.err.println("Barrier error: " + e.getMessage());
        }
    }
}
```

Listing 28: CyclicBarrier Thread Implementation

```
    public static void main(String[] args) {
            …
             try {
            if (opts.threads > 0) {
            ..
                                    // Option C: CyclicBarrier (uncomment to
    experiment - requires modification in MultiThreadWithBarrier)
                CyclicBarrier barrier = new CyclicBarrier(Math.min(opts.threads,
    opts.filenames.length) + 1);
                                  // Option C: enable/disable for CyclicBarrier
                  Thread multiThread = new MultiThreadWithBarrier(opts,
    fileCounter, opts.filenames.length, barrier);
                                   // Option C: CyclicBarrier
               barrier.await();
              System.out.println("Time taken (barrier):  " + (System.nanoTime()-
    starttime)/1000000000.0 + "s");
```

Listing 29: `main` method barrier implementation

**Sample Runs Output:**

Using 4 threads

```
-cp out cp3.lab04.crypt.JCrypt -d enigma -t 4 prac4-secrets/*.jpg.encrypted
..   ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken (latch): 7.94625425s
Using 4 threads for 8 files
Time taken (barrier):  10.451063583s
Using 4 threads for 8 files
Time taken (barrier):  9.608199917s
Using 4 threads for 8 files
```

Listing 30: Updated test script commands

Using 8 threads

```
java -cp out cp3.lab04.crypt.JCrypt -d enigma -t 8 prac4-secrets/*.jpg.encrypted
..   ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken (barrier):  8.658885s
Using 8 threads for 8 files
Time taken (barrier):  8.655880709s
Using 8 threads for 8 files
Time taken (barrier):  8.399285042s
Using 8 threads for 8 files
```

Listing 31: Updated test script commands

## 5.5 Lock Implementation

The `ReentrantLock` approach uses explicit locking to ensure thread-safe access to the shared file counter. This provides fine-grained control over the critical section where file indices are assigned.

```
static class MultiThreadWithLock extends Thread {
    private JCryptUtil.Options options;
    private int[] fileIndexCounter;
    private int totalFilesToProcess;
    private ReentrantLock fileAccessLock;

    public MultiThreadWithLock(JCryptUtil.Options opts, int[] counter, int
    totalFiles, ReentrantLock lock) {
        this.options = opts;
        this.fileIndexCounter = counter;
        this.totalFilesToProcess = totalFiles;
        this.fileAccessLock = lock;
    }

    @Override
    public void run() {
        int currentFileIndex;
        while (true) {
            fileAccessLock.lock();
            try {
                if (fileIndexCounter[0] ≥ totalFilesToProcess) {
                    break; // No more files to process
                }
                currentFileIndex = fileIndexCounter[0]++;
```

```
            } finally {
                fileAccessLock.unlock(); // Always release lock when done
            }

            try {
                process(options, currentFileIndex);
            } catch (JCryptUtil.Problem e) {
                System.err.println("ERROR in thread: " + e.getMessage());
            }
        }
    }
}
```

Listing 32: ReentrantLock Thread Implementation

```

    public static void main(String[] args) {
            …
             try {
            if (opts.threads > 0) {
            ..
                        // Option F: Thread Pool
                ExecutorService threadPool = Executors.newFixedThreadPool(
    numberOfThreads);

                // send individual file processings tasks to the pool
                for (int i = 0; i < opts.filenames.length; i++) {
                    final int fileIndex = i;
                    threadPool.submit(() →  {
                        try {
                            process(opts, fileIndex);
                        } catch (JCryptUtil.Problem e) {
                            System.err.println("ERROR in thread: " + e.getMessage()
    );}
                    });
                }
                threadPool.shutdown();
                try {
                    threadPool.awaitTermination(Long.MAX_VALUE, TimeUnit.
    NANOSECONDS);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Time taken (Thread Pool): " + (System.nanoTime
    ()-starttime)/1000000000.0 + "s");
```

Listing 33: `main` method Java's thread pooling implementation

**Sample Runs Output:**

Using 4 threads

```
-cp out cp3.lab04.crypt.JCrypt -d enigma -t 4 prac4-secrets/*.jpg.encrypted
..   ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h#c'    p7_  R ?
Time taken: 8.53083775s
Using 4 threads for 8 files
Time taken: 8.275500709s
Using 4 threads for 8 files
Time taken: 10.14398025s
```

```
Using 4 threads for 8 files
```

Listing 34: Updated test script commands

Using 8 threads

```
java -cp out cp3.lab04.crypt.JCrypt -d enigma -t 8 prac4-secrets/*.jpg.encrypted
..   ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken: 9.7056815s
Using 8 threads for 8 files
Time taken: 8.764813042s
Using 8 threads for 8 files
Time taken: 10.31640575s
Using 8 threads for 8 files
```

Listing 35: Updated test script commands

### 5.6 Thread Pooling Implementation

This approach uses Java's built-in thread pool framework. Tasks are submitted to a fixed-size thread pool, and `awaitTermination()` makes sure all tasks complete before proceeding.

```
    ExecutorService threadPool = Executors.newFixedThreadPool(numberOfThreads);

                // send individual file processings tasks to the pool
                for (int i = 0; i < opts.filenames.length; i++) {
                    final int fileIndex = i;
                    threadPool.submit(() →  {
                        try {
                            process(opts, fileIndex);
                        } catch (JCryptUtil.Problem e) {
                            System.err.println("ERROR in thread: " + e.getMessage()
    );
                        }
                    });
                }

                threadPool.shutdown();
                try {
                    threadPool.awaitTermination(Long.MAX_VALUE, TimeUnit.
    NANOSECONDS);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Time taken (Thread Pool): " + (System.nanoTime
    ()-starttime)/1000000000.0 + "s");
```

Listing 36: ExecutorService Thread Pool Implementation

**Sample Runs Output:** Using 4 threads

```
-cp out cp3.lab04.crypt.JCrypt -d enigma -t 4 prac4-secrets/*.jpg.encrypted
..   ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken (Thread Pool): 16.967400292s
Using 4 threads for 8 files
Time taken (Thread Pool): 16.017159583s
Using 4 threads for 8 files
Time taken (Thread Pool): 20.123930584s
```

```
Using 4 threads for 8 files
```

Listing 37: Updated test script commands

Using 8 threads

```
java -cp out cp3.lab04.crypt.JCrypt -d enigma -t 8 prac4-secrets/*.jpg.encrypted
..    ioTZ[   I0kbLA  ~   Sw@Hsg#    i37X2r  #h # c'    p7_  R ?
Time taken (Thread Pool): 17.189250666s
Using 8 threads for 8 files
Time taken (Thread Pool): 19.813550708s
Using 8 threads for 8 files
Time taken (Thread Pool): 17.922120206s
Using 8 threads for 8 files
```

Listing 38: Updated test script commands

# 6 References


[1] Gen AI LLM Assistant (Explanations, Coding Questions, Idea Generation and Planning, Code Comments, Debugging, Report Planning, Document Formatting). Google, Gemini 2.5 Pro Preview [06-05-2025].

[2] GeeksforGeeks. "Thread Pools in Java." https://www.geeksforgeeks.org/thread-pools-java/. Accessed June 2025.

[3] Tuan H. "Java Thread Pool – How to Efficiently Manage Threads." *Medium*, https://medium.com/tuanhdotnet/java-thread-pool-how-to-efficiently-manage-threads-4fb8c1edb113. Accessed June 2025.

[4] Sainath Batthala. "Java Multi-threading: CyclicBarrier Use Case and Example." *Medium*, https://medium.com/@sainathbatthala/java-multi-threading-cyclicbarrier-use-case-and-example-6958d5a945a2. Accessed June 2025.

[5] Alexander Obregon. "Java's CountDownLatch await() Method Explained." *Medium*, https://medium.com/@AlexanderObregon/javas-countdownlatch-await-method-explained-2fa0c0c4b044. Accessed June 2025.

[6] Baeldung. "Java CountDownLatch." https://www.baeldung.com/java-countdown-latch. Accessed June 2025.

[7] GeeksforGeeks. "ReentrantLock in Java." https://www.geeksforgeeks.org/reentrant-lock-java/. Accessed June 2025.

[8] JavaCodeGeeks. "Java Concurrency Cheatsheet." https://www.javacodegeeks.com/java-concurrency-cheatsheet.html. Accessed June 2025.