発展プログラミング

第9回：マルチスレッドによる処理の高速化の基礎

宮田章裕 <miyata.akihiro@nihon-u.ac.jp>

前回講義の復習

スレッドの待ち合わせ (1/3)

- 単純な実装では各スレッドが独立して動いてしまう
  - Thread2・3終了後に，Thread1で特定の処理を実行，ということができない

MyThread1.java

```java
public class MyThread1 implements Runnable {

    private final static long SLEEP_LEN_MSEC = 1000;

    private int id;


    public MyThread1(int id) {
        this.id = id;
    }

    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println("MyThread #" + id + ": " + i);
            try {
                Thread.sleep(SLEEP_LEN_MSEC);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

C

Main.java

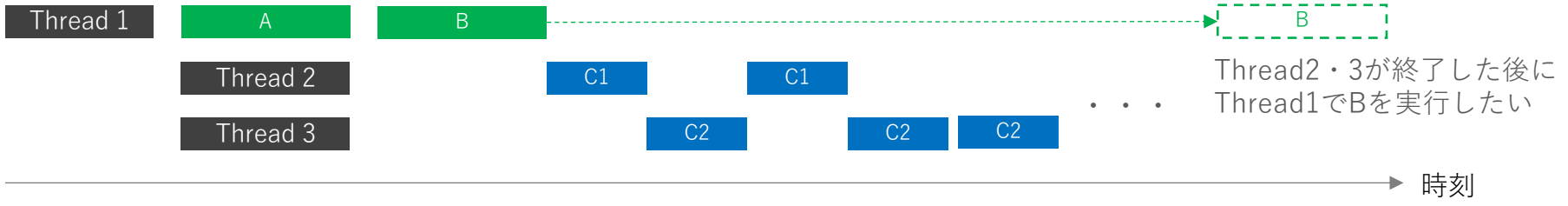
```java
public class Main {

    private final static int THREAD_COUNT = 2;

    public static void main(String[] args) {
        for(int i = 0; i < THREAD_COUNT; i++) {
            MyThread1 mt = new MyThread1(i);
            Thread thread = new Thread(mt);
            thread.start();
        }

        System.out.println("FINISH");
    } Bが最後に実行されることを
}     期待するが，実際はそうならない
```

A

B

実行結果

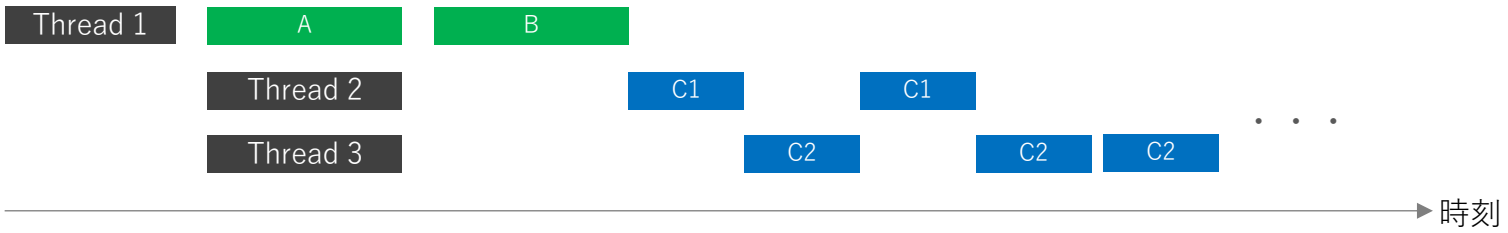
```
FINISH
MyThread #1: 0
MyThread #0: 0
MyThread #1: 1
MyThread #0: 1
MyThread #1: 2
MyThread #0: 2
MyThread #1: 3
MyThread #0: 3
MyThread #1: 4
MyThread #0: 4
MyThread #1: 5
MyThread #0: 5
MyThread #1: 6
MyThread #0: 6
MyThread #1: 7
MyThread #0: 7
MyThread #1: 8
MyThread #0: 8
MyThread #1: 9
MyThread #0: 9
```



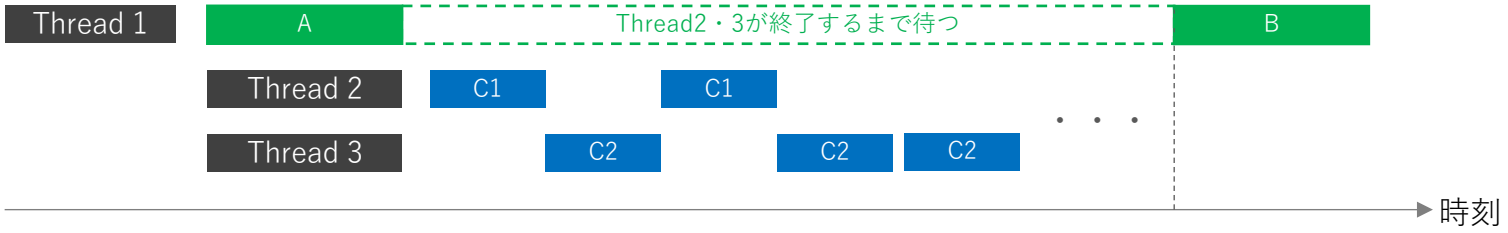
スレッドの待ち合わせ (2/3)

- join()で特定のスレッドの終了を待つことができる

join()を使わない場合（各スレッドが独立して動く）



Thread1でjoin()を使う場合（Thread2・3の終了を待ってからThread1が動く）



スレッドの待ち合わせ (3/3)

Counter.java

```java
public class Counter implements Runnable {

    private final static long SLEEP_LEN_MSEC = 1000;

    private int id;

    public Counter(int id) {
        this.id = id;
    }

    public void run() {
        for(int i = 0; i < 5; i++) {
            System.out.println("Counter #" + id + ": " + i);
            try {
                Thread.sleep(SLEEP_LEN_MSEC);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Main.java

```java
public class Main {

    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter(0);
        Thread thread = new Thread(c);
        thread.start();

        thread.join();

        System.out.println("FINISH");
    }

}
```

thread.join()の実行に必要

ここで，threadインスタンスで
開始したスレッドが終了するのを待つ
（この行をコメントアウトして挙動の変化を確認してみよう）

実行結果

```
Counter #0: 0
Counter #0: 1
Counter #0: 2
Counter #0: 3
Counter #0: 4
FINISH
```

狙い通り，他スレッドが終了した後で
System.out.println(“FINISH”); が実行される

共有資源へのアクセス (1/3)

- 各スレッドは共有資源にアクセスできる

Shared.java

```java
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

}
```

MyThread.java

```java
public class MyThread implements Runnable {

    private Shared shared;

    public MyThread(Shared shared) {
        this.shared = shared;
    }

    public void run() {
        System.out.println(shared.getValue());
    }

}
```

Main.java

```java
public class Main {

    public static void main(String[] args) {

        Shared shared = new Shared();

        Thread[] threads = new Thread[3];
        for(int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new MyThread(shared));
            threads[i].start();
        }
    }

}
```

同じSharedクラスの
インスタンスを
各スレッドに渡している

実行結果 (3つのスレッドは同じ値を取得できる)

```
0
0
0
```

共有資源へのアクセス (3/3)

- 各スレッドは共有資源を独自のタイミングで利用してしまう

Shared.java

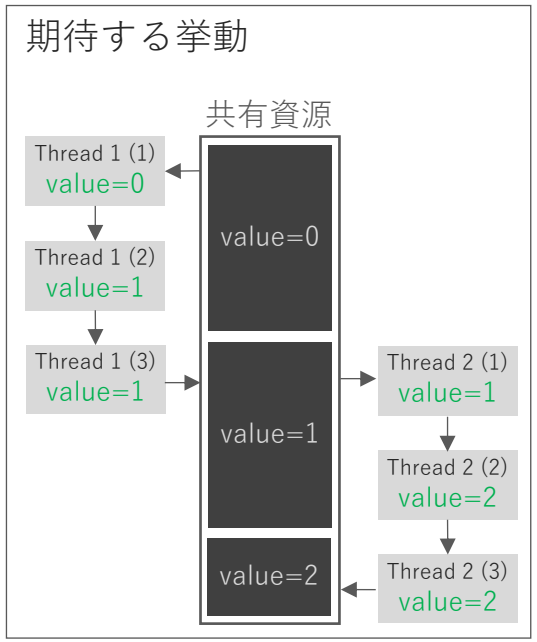
```java
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

    public void changeValue() {
        value++;
    }

}
```

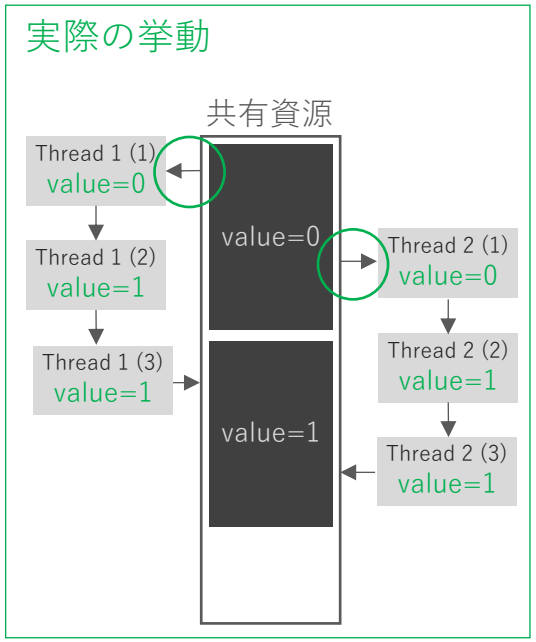
MyThread.java

```java
package shared;

public class MyThread implements Runnable {

    private Shared shared;

    public MyThread(Shared shared) {
        this.shared = shared;
    }

    public void run() {
        for(int i = 0; i < 10000; i++) {
            shared.changeValue();
        }
    }

}
```

各スレッドがshared.changeValue()を実行するとき
具体的には下記ステップが実行されている

(1) 自スレッドのメモリにvalueの値をコピー
(2) 自スレッドのメモリでvalueの値を増やす
(3) (2)を共有インスタンスのvalueの値に反映





排他制御 (1/3)

Synchronization

- 共有資源の不整合を回避する手段
- 特定コードを同時に実行できるスレッドを1つに限定する

```
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

    public void changeValue() {
        value++;
    }
}
```

この部分を同時に実行できるスレッドを
1つに限定できれば不整合が生じない

排他制御 (2/3)

Synchronization

- 方法1：Synchronized statements
  - 1スレッドだけに実行させたい範囲をsynchronizedで囲む
  - その範囲を実行するためのロックを指定する
  - ロックは全スレッド中で唯一のものである必要があり，典型的には
    (1) その範囲を実行するのに必要なインスタンスか
    (2) 専用に用意したロック用のObject型のインスタンス変数を用いる

(1)の書き方

```
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

    public void changeValue() {

        synchronized(this) {
            value++;
        }

    }
}
```

(2)の書き方

```
public class Shared {

    private Object lock = new Object();

    private int value = 0;

    public int getValue() {
        return value;
    }

    public void changeValue() {

        synchronized(lock) {
            value++;
        }

    }
}
```

排他制御 (3/3)

Synchronization

- 方法2：Synchronized method
  - 1スレッドだけに実行させたいメソッドにsynchronizedの指定を行う

```
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

    public synchronized void changeValue() {
        value++;
    }
}
```

デッドロック

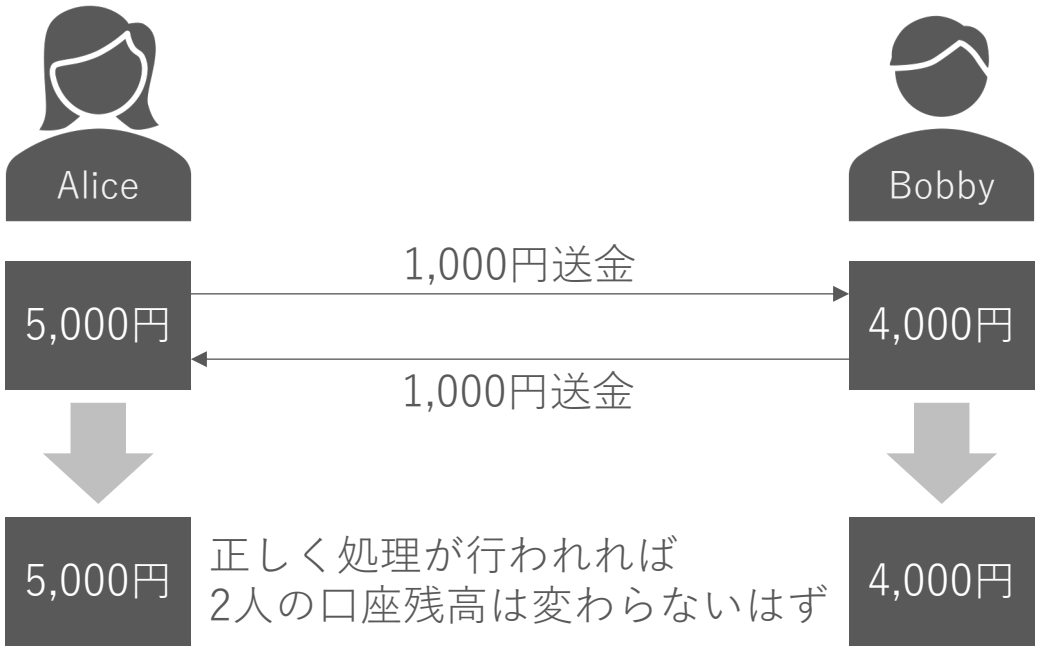
作業準備

- 本日の作業ディレクトリの作成・移動
  - mkdir -p SOMEWHERE/2021_ap/09
  - 以降，SOMEWHERE/2021_ap/09をWORK_DIRとする

- 作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 09: Multithreading 3 > Code > synchronized_test.zip をWORK_DIRにダウンロード
  - unzip synchronized_test.zip
  - cd synchronized_test

口座送金問題

- 2つの口座間で送金を行うシーンを考える
  - Aliceの口座から，Bobbyの口座へ，1,000円送金する
  - Bobbyの口座から，Aliceの口座へ，1,000円送金する



IR09-1

- synchronized_test内の下記プログラムは，Aliceの口座からBobbyの口座に1,000円，Bobbyの口座からAliceの口座に1,000円の送金を並行して行うことを意図したものである。各口座で不整合が生じないよう送金時（TransferThreadクラスのrun()内）には排他制御も行っている。このプログラムを理解した上で，実行せよ。すると，何も標準出力されないはずなので，その結果が生じた理由を推測せよ。

Account.java

```java
public class Account {

    private String owner;
    private int number;
    private int balance;

    public Account(String owner,
                   int number,
                   int balance) {
        this.owner = owner;
        this.number = number;
        this.balance = balance;
    }

    public void deposit(int amount) {
        balance += amount;
    }

    public void withdraw(int amount) {
        balance -= amount;
    }

    public int getNumber() {
        return number;
    }

    public void displayBalance() {
        System.out.println(owner
                           + "'s balance: "
                           + balance);
    }

}
```

TransferThread.java

```java
public class TransferThread implements Runnable {

    private final static int TRANSFER_COUNT  = 1000;
    private final static int TRANSFER_AMOUNT = 1;

    private Account fromAccount;
    private Account toAccount;

    public TransferThread(Account fromAccount,
                          Account toAccount) {
        this.fromAccount = fromAccount;
        this.toAccount   = toAccount;
    }

    public void run() {
        for(int i = 0; i < TRANSFER_COUNT; i++) {
            synchronized(fromAccount) {
                synchronized(toAccount) {
                    fromAccount.withdraw(TRANSFER_AMOUNT);
                    toAccount.deposit(TRANSFER_AMOUNT);
                }
            }
        }
    }
}
```

Main.java

```java
public class Main {

    public static void main(String[] args)
        throws InterruptedException {
        Account aliceAccount = new Account("Alice",
                                           100,
                                           5000);
        Account bobbyAccount = new Account("Bobby",
                                           200,
                                           4000);


        Thread a2bThread
            = new Thread(
                  new TransferThread(aliceAccount,
                                     bobbyAccount));
        Thread b2aThread
            = new Thread(
                  new TransferThread(bobbyAccount,
                                     aliceAccount));

        a2bThread.start();
        b2aThread.start();

        a2bThread.join();
        b2aThread.join();

        aliceAccount.displayBalance();
        bobbyAccount.displayBalance();
    }

}
```

並行処理を含むアプリケーションの開発

予告

業外レポート OR10 (1/2)

❖課題：並行処理を含むアプリケーションを開発し，その説明書を作成せよ。

❖提出物

- アプリケーション（zip形式，ファイル名：541xxxx_app.zip）
  - Javaで実装すること。
  - 並行処理を含むこと。
  - mainメソッドを含むクラスをMainという名称にすること。
  - アプリケーション動作に必要な全てのファイル・ディレクトリを上記ファイル名のzipファイルにひとまとめにすること。
- ❖ 説明書（PDF形式，ファイル名：541xxxx_doc.pdf）
  - 「アプリケーション名」，「想定シーン」，「並行して行う処理とその理由」，「実装上の制約」，「クラス・メソッドの一覧と説明」，「並行処理を実現するために工夫した点」を独立した項目として明記し，各内容を記載すること。
  - 「想定シーン」には実際に行っていない処理（例：口座への送金，動画のダウンロード）を含めて書いて良い。ただし，どの処理を実際には行っていないのか「実装上の制約」に明記すること。実際に行っていない処理はダミーコード（例：sleep）で表現して良い。

予告

業外レポート OR10 (2/2)

- 採点基準
  - オリジナリティ (70%)
  - 内容の妥当性 (30%)

- 提出期限・提出方法
  - 12/27(月) 23:59
  - Bbの指定ページに前ページで指定したファイルを提出する。
  - 提出期限を過ぎると提出場所が消える。

- 諸注意
  - これは「レポート」である。メモ書きのような，レポートの体裁をなしていないものは採点対象外とする。
  - 他者に伝わるような，適切な文章で記述すること。これには入念な下書き・推敲が要るはずである。
  - 他者のコードの盗用が発覚した場合は，厳重に処罰する。学生間で類似レポートが発見された場合は，見せた方・見た方を問わず，双方を採点対象外・処罰の対象とする。
  - 講義資料中のプログラムと同等とみなせるもの（例：変数名を変えただけ）は採点対象外。

並行処理を含むアプリケーションの開発

- アプリケーション名：Document Analyzer
- 想定シーン：大量のドキュメントをDLして分析する

ベースとなるアプリケーション (1/6)

- 想定シーン：大量のドキュメントをDLして分析する
- 実装上の制約

本来の実装	代用の実装
大量のドキュメント	1つのテキストファイル
1つのドキュメント	上記テキストファイルの1行
ドキュメントのダウンロード	ローカルディスクからのファイル読み込み
ドキュメントの分析	行中の単語数のカウント
時間がかかる処理	sleep

ベースとなるアプリケーション (2/6)

❖プログラムのアウトライン

DocDownloader.java

```
public class DocDownloader {

    public void start() {
        全ドキュメントをダウンロードする。
    }

    public String getLine() {
        先頭のドキュメントを返す。
    }

}
```

Main.java

```
public class Main {

    public static void main(String[] args) {
        DocDownloaderで全ドキュメントをダウンロードする。
        WordCounterで全ドキュメントを分析する。
        Resultから分析結果を取得して出力する。
    }

}
```

WordCounter.java

```
public class WordCounter {

    public void start() {
        DocDownloaderから
        1つずつドキュメントを取得して分析する。
        分析結果をResultに反映する。
    }

}
```

Result.java

```
public class Result {

    public void addWordCount(int count) {
        分析結果を更新する。
    }

    public int getWordCount() {
        分析結果を取得する。
    }
}
```

ベースとなるアプリケーション (3/6)

❖ DocDownloader

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.stream.Stream;

public class DocDownloader {
    private final static long DOWNLOAD_TIME_MSEC = 100;

    private Deque<String> deque = new ArrayDeque<String>();
    private String docFile;

    public DocDownloader(String docFile) {
        this.docFile = docFile;
    }

// To top right.
```

```
public interface Deque<E>
extends Queue<E>

A linear collection that supports element insertion and removal at both ends.
The name deque is short for "double ended queue" and is usually pronounced
"deck". Most Deque implementations place no fixed limits on the number of
```

https://docs.oracle.com/javase/10/docs/api/java/util/Deque.html

docFileの中身を1行ずつdequeの末尾に追加する処理。
DLにかかる時間をシミュレートするために
sleep処理を行っている。

```java
// From bottom left.

    public void start() {
        Stream<String> stream = null;
        try {
            stream = Files.lines(Paths.get(docFile));
        } catch (IOException e) {
            e.printStackTrace();
        }

        stream.forEach(line -> {
            try {
                Thread.sleep(DOWNLOAD_TIME_MSEC);
            } catch(InterruptedException e) {
                e.printStackTrace();
            }
            deque.addLast(line);
            System.out.println("DocDownloader: " + line);
        });

        System.out.println("DocDownloader: FINISHED");
    }

    public String getLine() {
        if(deque.isEmpty()) {
            return null;
        } else {
            return deque.removeFirst();
        }
    }
}
```

dequeが空ならnullを返す。
dequeに要素が入っているなら，
先頭要素を返した上で，当該要素をdequeから削除。

ベースとなるアプリケーション (4/6)

Result

```java
public class Result {
    private final static long ADD_TIME_MSEC = 500;

    private int wordCount;

    public Result() {
        wordCount = 0;
    }

    public void addWordCount(int count) throws InterruptedException {
        Thread.sleep(ADD_TIME_MSEC);
        wordCount += count;
    }

    public int getWordCount() {
        return wordCount;
    }
}
```

与えられた数（count）を
合計数（wordCount）に加算する処理。
加算処理にかかる時間をシミュレートするために
sleep処理を行っている。

ベースとなるアプリケーション (5/6)

WordCounter

```
public class WordCounter {
    private final static long ANALYSIS_TIME_MSEC = 500;

    private DocDownloader downloader;
    private Result result;

    public WordCounter(DocDownloader downloader, Result result) {
        this.downloader = downloader;
        this.result = result;
    }

    private int countWord(String string) {
        String[] words = string.split("\\s+");
        int count = words.length;
        return count;
    }

    public void start() {
        String line = downloader.getLine();
        while(line != null) {
            try {
                Thread.sleep(ANALYSIS_TIME_MSEC);
                int count = countWord(line);
                System.out.println("\tWordCounter: " + line + " -> " + count);
                result.addWordCount(count);
                line = downloader.getLine();
            } catch(InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("\tWordCounter: FINISHED");
    }
}
```

英文文字列を空白位置で分割して得られる
各要素（＝各単語）をString配列に格納し，
配列長（＝単語数）を求める処理。

downloader.getLine()で各行を取得し，
各行の単語数をresultに反映する処理。
downloader.getLine()がnullを返したら終了。
分析にかかる時間をシミュレートするために
sleep処理を行っている。

ベースとなるアプリケーション (6/6)

Main

```
public class Main {
    private final static String DOC_FILE = "short.txt";

    public static void main(String[] args) throws InterruptedException {
        long startTimeMsec = System.currentTimeMillis();

        DocDownloader downloader = new DocDownloader(DOC_FILE);
        downloader.start();

        Result result = new Result();
        WordCounter wordCounter = new WordCounter(downloader, result);
        wordCounter.start();

        System.out.println("Main: " + result.getWordCount() + " words");

        long endTimeMsec = System.currentTimeMillis();
        System.out.println("Processing time: " + (endTimeMsec - startTimeMsec) + " msec");
    }
}
```

DocDownloaderの
インスタンスを作成して
ダウンロードを実行して
完了を待つ。

WordCounterの
インスタンスを作成して
分析を実行して
完了を待つ。

作業準備

- 作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 09: Multithreading 3 > Code > doc_analyzer.zip をWORK_DIRにダウンロード
  - unzip doc_analyzer.zip
  - cd doc_analyzer

IR09-2

- doc_analyzer内の解説・コードを読み込んで理解を深め，実際に動作させて挙動を確認せよ。その後，どの処理とどの処理を並行して行えば処理が効率化できるか検討せよ。

実行結果

```
DocDownloader: Alan Mathison Turing OBE FRS (23 June 1912 – 7 June 1954) was an
DocDownloader: English mathematician, computer scientist, logician, cryptanalyst,
DocDownloader: philosopher and theoretical biologist. Turing was highly influential in the
.
.
.
DocDownloader: that this work shortened the war in Europe by more than two years and saved
DocDownloader: over 14 million lives.
DocDownloader: FINISHED
      WordCounter: Alan Mathison Turing OBE FRS (23 June 1912 – 7 June 1954) was an -> 14
      WordCounter: English mathematician, computer scientist, logician, cryptanalyst, -> 6
      WordCounter: philosopher and theoretical biologist. Turing was highly influential in the -> 10
         .
         .
         .
      WordCounter: that this work shortened the war in Europe by more than two years and saved -> 15
      WordCounter: over 14 million lives. -> 4
      WordCounter: FINISHED
Main: 263 words
Processing time: 26649 msec
```

ダウンロード処理と分析処理の並行化

DocDownloader.java

```
public class DocDownloader {

    public void start() {
        全ドキュメントをダウンロードする。
    }

    public String getLine() {
        先頭のドキュメントを返す。
    }

}
```

Main.java

```
public class Main {

    public static void main(String[] args) {
        DocDownloaderで全ドキュメントをダウンロードする。
        WordCounterで全ドキュメントを分析する。
        Resultから分析結果を取得して出力する。
    }

}
```

WordCounter.java

```
public class WordCounter {

    public void start() {
        DocDownloaderから
        1つずつドキュメントを取得して分析する。
        分析結果をResultに反映する。
    }

}
```

1つのドキュメントをDLする（実際は，ファイルの1行を読み込む）たびに
NW遅延等（実際は，DOWNLOAD_TIME_MSECのsleep）の待ち時間が生じている。

DL時の待ち時間中に，
DL済みのドキュメント（実際は，読み込み済みの行）の
分析処理を進めれば処理全体を高速化できる。

具体的な実装は次回行う

本日のまとめ

❖講義内容

- デッドロック
- 並行処理を含むアプリケーションの開発

❖授業内課題提出

- 各授業内課題（IR）の解答を記載せよ。
- 「講義内容のまとめ」の解答欄に
  上記「講義内容」の各項目について文章で説明を記載せよ。