SMU
SINGAPORE MANAGEMENT UNIVERSITY

School of Computing and Information Systems

[CS205] Operating Systems Concepts with Android

Lab #2: Java Concurrency

Prerequisites

Install JDK and IDE

Before class, please make sure to install a recent version of Java JDK to actively participate in the lab. Visit the Oracle's Java Downloads website for more details.

For best experience, install an IDE or a plain-text editor of your choice, e.g. VSCode, IntelliJ.

Refresh Java knowledge

To focus on newly introduced elements of Java programming language that are related to multithreading and concurrency and efficiently put them to use, make sure that your proficiency in basic syntax and semantics of Java programming has been brought up-to-date.

Creating threads

Every Java application has at least one thread, the "*main thread*". Additional threads can be created from the main thread on demand. In Java, there are multiple ways of achieving this goal.

Extending a class `Thread`

- One way is to extend the `Thread` class and override the `run()` method.
- To spawn a new thread, instantiate an object of the `Thread` class. As new threads are initially suspended, to begin its execution call the `start()` method. This will invoke the `run()` method of the thread.
- Note that without additional synchronization, there is no guarantee which threads run first.
- You can call the `join()` method to make a calling thread wait for the thread whose method was called to finish running before merging its flow of execution with the calling thread.

Code example

```java
public class ThreadExample {

    public static void main(String[] args) {
        final String threadName = Thread.currentThread().getName();
        System.out.println(threadName + " is now running");

        final MyThread thread1 = new MyThread();
        final MyThread thread2 = new MyThread();

        thread1.start();
        thread2.start();

        try {
            thread1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Done!");
    }
}

class MyThread extends Thread {

    @Override
    public void run() {
        final String threadName = Thread.currentThread().getName();
        System.out.println(threadName + " is now running");
    }
}

```

Compilation and execution

```
javac -d ../bin ThreadExample.java && java -cp ../bin ThreadExample
```

Output

Typical output

```
main is now running
Thread-0 is now running
Thread-1 is now running
Done!
```

Possible alternative output

```
main is now running
Thread-1 is now running
Thread-0 is now running
Done!
```

Implementing an interface `Runnable`

- An alternate way to create threads is to implement the `Runnable` interface through an anonymous class declaration, and override its method `run()`. A constructor instantiating a new `Thread` object accepts a `Runnable` object as a parameter.
- Note that you could create one `Runnable` object and pass it to multiple threads. For example, you could define a task with a `Runnable` instance, and then create multiple threads that perform this same task. Typically you would use one `Runnable` to represent one task.

Code example

```java
public class RunnableExample {

    public static void main(String[] args) {
        final Runnable runnable = new Runnable() {
            @Override
            public void run() {
                String thread_name = Thread.currentThread().getName();
                System.out.println(thread_name + " is running");
            }
        };

        final Thread thread1 = new Thread(runnable, "first thread");
        final Thread thread2 = new Thread(runnable, "second thread");

        thread1.start();
        thread2.start();

        try {
            thread1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Done!");
    }
}

```

Compilation and execution

```
javac -d ../bin RunnableExample.java && java -cp ../bin RunnableExample
```

Output

Typical output

```
first thread is running
second thread is running
Done!
```

Possible alternative output

```
second thread is running
first thread is running
Done!
```

Using a lambda expression

- A lambda expression is a high-level programming feature supported by Java (and many other programming languages) that offers a shorthand syntax for implementing classes that carry behaviour, rather than data. Since to a constructor of a `Thread` class we want to pass a `Runnable` logic to be executed by that thread, a lambda expression seems to be a perfect candidate in many scenarios due to its brevity.
- The syntax of a lambda expression allows for single expressions and statement blocks:

  ```
  // () -> expression statement
  () -> a + b
  ```

  ```
  // () -> statement block
  () -> {
      return a + b;
  }
  ```

  Take note that the `return` statement is automatically implied in a lambda expression with an *expression statement*, and only required if a lambda expression with a *statement block* produces a result.
- In Java, depending on arguments (that can be included in `()`) and a type of a returned result object, a lambda expression is resolved into an anonymous functor class that automatically implements one of the following interfaces:
  - Supplier - no arguments, returns a result; `() -> { return x; }`.
  - Callable - no arguments, returns a result, and may throw; `() -> { return x; }`.
  - Consumer - one argument `x` of type `T`, returns `void`; `(x) -> {}`.
  - Runnable - no arguments, returns `void`; `() -> {}`.

  The last one, `Runnable`, is of particular interest to us when dealing with multithreaded applications as it shortens definitions of threads' functions.

Code example

```java
public class LambdaExample {

    public static void main(String[] args) {
        final Thread thread1 = new Thread(() -> {
            System.out.println("first thread is running");
        });
        final Thread thread2 = new Thread(() -> {
            System.out.println("second thread is running");
        });

        thread1.start();
        thread2.start();

        try {
            thread1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Done!");
    }
}

```

Compilation and execution

```
javac -d ../bin LambdaExample.java && java -cp ../bin LambdaExample
```

Output

Typical output

```
first thread is running
second thread is running
Done!
```

Possible alternative output

```
second thread is running
first thread is running
Done!
```

Working with threads

The `volatile` keyword

- When threads access a variable, each thread will store the value of this variable in a thread-specific context, for example associated CPU registers or cache memory, before manipulation. In the Java applications, this memory may correspond to the heap or the stack areas of the Java Virtual Machine. As a consequence, there is no guarantee that each thread will see the most updated value of the variable.
- To ensure threads see the same value of a variable and access it directly from the main memory, we need to qualify the variable as `volatile`. This ensures that whenever a write operation is carried out, the update is flushed to the main memory; and whenever a read operation is carried out, the value is obtained directly from the main memory.
- Other programming languages often offer a keyword with a similar role (C, C++, and C# use keyword `volatile`), but due to their less abstract memory model the underlying behaviour offered by this keyword may slightly differ.

Code example

```java
public class VolatileExample {

    private static volatile boolean done = false;
    // private static boolean done = false;

    private static void sleep(int n) {
        try {
            Thread.sleep(n);
        } catch (InterruptedException e) {
        }
    }

    public static void main(String[] args) {
        final Thread thread1 = new Thread(() -> {
            while (!done);
            System.out.println("Done!");
        });

        final Thread thread2 = new Thread(() -> {
            done = true;
        });

        thread1.start();
        sleep(100);
        thread2.start();

        try {
            thread1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            thread2.join();
```

```java
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Compilation and execution

```
javac -d ../bin VolatileExample.java && java -cp ../bin VolatileExample
```

Output

The original program prints the text:

```
Done!
```

However, the program where `done` is not `volatile` will hang, as the changes to the value of the variable are not visible across threads.

Locks and the `synchronized` keyword

- To coordinate running of different threads for mutual exclusion, we use a lock. To create a lock, simply instantiate any Java object, even simply `Object`. The reason is that every Java object is associated with an implicit lock.
- Use the `synchronized` keyword around a block of code to indicate that it represents a critical section. Java guarantees that no two threads can enter the critical sections, synchronized on a given lock, at any one time.

Code example

```java
public class LockExample {

    private static final long maxElement = 100_000;

    private static final int threadCount = 2;

    private static final Object lock = new Object();

    private static long sum = 0;

    public static void main(String[] args) {
        final Runnable adder = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < maxElement; i++) {
                    // sum++;
                    synchronized(lock) {
                        sum++;
                    }
                }
            }
        };
        final Thread[] threads = new Thread[threadCount];
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new Thread(adder);
            threads[i].start();
        }
        for (int i = 0; i < threadCount; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("sum = " + sum);
    }
}

```

Compilation and execution

```
javac -d ../bin LockExample.java && java -cp ../bin LockExample
```

Output

The original program prints the output below. However, the program where the `sum` variable is updated without a lock may produce the output that does not include arbitrary elements added to `sum` due to a *race condition*.

```
sum = 200000
```

Condition variables, `await()` and `signal()` methods

- In many scenarios, you want a thread to execute only after another thread completes its task. To synchronize this, we can use a condition variable.
- You first create an explicit `Lock`, and use the method `newCondition()` to create a condition variable. Note that you can create multiple condition variables associated with the same lock instance.
- A condition variable is associated with the waiting queue. Use `await()` to put the thread into the queue, and `signal()` to wake up a waiting thread from the queue.

Code example

Note that the `mutexLock()` method has been implemented using an interface `Consumer<T>` (see the section Using a lambda expression for details). This implementation is not required, but makes it apparent that a locked mutex always gets unlocked.

```
import java.util.concurrent.Callable;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
import java.util.function.Consumer;

public class CondVarExample {

    private static final Lock mutex = new ReentrantLock();

    private static final Condition condNumDone = mutex.newCondition();

    private static final Condition condCharDone = mutex.newCondition();

    private static volatile char buffer[] = new char[20];

    private static volatile int index = 0;

    private static void randomSleep() {
        try {
            final int n = (int)(Math.random() * 10);
            Thread.sleep(n);
        } catch (InterruptedException e) {
        }
    }

    private static void mutexLock(Consumer<Integer> action, int i) {
        mutex.lock();
        try {
            action.accept(i);
        } finally {
            mutex.unlock();
        }
    }

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

```java
        final Thread numThread = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                // buffer[index++] = (char)('0' + i);
                // randomSleep();
                mutexLock((ii) -> {
                    while (index % 2 == 1) {
                        /* Release the lock and wait until another thread
                           calls condCharDone.signal */
                        try {
                            condCharDone.await();
                        } catch (InterruptedException e) {
                        }
                    }
                    buffer[index] = (char)('0' + ii);
                    index++;
                    randomSleep();
                    condNumDone.signal();
                }, i);
            }
        });

        final Thread charThread = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                // buffer[index++] = (char)('A' + i);
                // randomSleep();
                mutexLock((ii) -> {
                    while (index % 2 == 0) {
                        /* Release the lock and wait until another thread
                           calls condNumDone.signal */
                        try {
                            condNumDone.await();
                        } catch (InterruptedException e) {
                        }
                    }
                    buffer[index] = (char)('A' + ii);
                    index++;
                    randomSleep();
                    condCharDone.signal();
                }, i);
            }
        });

        final Thread[] threads = {
            numThread,
            charThread
        };

        for (Thread thread : threads) {
            thread.start();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
                }
            }

            System.out.println(buffer);
        }
    }
}
```

Compilation and execution

```
javac -d ../bin CondVarExample.java && java -cp ../bin CondVarExample
```

Output

```
0A1B2C3D4E5F6G7H8I9J
```

Producer-Consumer with a monitor

- In Java under-the-hood all objects are associated implicitly with a monitor, which is essentially a lock with a condition variable. Hence a monitor gives you mutual exclusion and signaling mechanisms.
- Here is an example illustrating how to use a monitor to solve a variant of the Producer-Consumer problem.
    - A producer makes `n` hotdogs, and puts them onto a circular queue buffer.
    - A consumer that takes hotdogs (and pack them) if available in the queue, on the first-in/first-out basis.
    - The queue size is `size`, where `size` < `n`. As such, the producers can only put the hotdogs in the queue if there is available capacity.
- Note the following synchronization mechanisms:
    - We use the keyword `synchronized` on the `put()` and `get()` methods belonging to the same class `Buffer`. This means that when the `put()` method acquires the lock for the `Buffer` object, the caller of the `get()` method will not be able to run it immediately and will have to wait, and vice versa.
    - In the `put()` method, a producer checks if buffer is full. If it is, it waits until the buffer is not full again – this is achieved by releasing the lock first so that the some items can be consumed via the `get()` method. Once buffer is not full again, the producer acquires the lock, puts an item on queue, then notifies all threads.
    - Similarly, in the `get()` method, a consumer checks if buffer is empty. If it is, it waits until the buffer is not empty again – this is achieved by releasing the lock first so that the some items can be produced via the `put()` method. Once the buffer is not empty again, the consumer acquires the lock, gets an item from queue, then notifies all threads.

Code example

```
public class ProducerConsumer {

    private static final long limit = 300_000_000;

    private static int n;

    private static int size;

    private static void doWork(int n) {
        for (int i = 0; i < n; i++) {
            long m = limit;
            while (m > 0) {
                m--;
            }
        }
    }

    public static void main(String[] args) {
        n = Integer.parseInt(args[0]);
        size = Integer.parseInt(args[1]);
        final Buffer buffer = new Buffer(size);

        final Thread producer = new Thread(() -> {
            for (int i = 0; i < n; i++) {
                doWork(2);
                final Hotdog hotdog = new Hotdog(i);
                buffer.put(hotdog);
            }
        });

        final Thread consumer = new Thread(() -> {
            for (int i = 0; i < n; i++) {
                @SuppressWarnings("unused")
                final Hotdog hotdog = buffer.get();
                doWork(5);
            }
        });

        final Thread[] threads = {
            producer,
            consumer
        };

        for (Thread thread : threads) {
            thread.start();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```java
        }
    }
}

class Buffer {

    private static volatile Hotdog[] buffer;

    private static volatile int front = 0;

    private static volatile int back = 0;

    private static volatile int itemCount = 0;

    Buffer(int size) {
        buffer = new Hotdog[size];
    }

    synchronized void put(Hotdog hotdog) {
        while (itemCount == buffer.length) {
            try {
                this.wait();
            } catch (InterruptedException e) {
            }
        }
        buffer[back] = hotdog;
        back = (back + 1) % buffer.length;
        System.out.println(
            "Item count: " + itemCount + ", " +
            "Producing " + hotdog
        );
        itemCount++;
        this.notifyAll();
    }

    synchronized Hotdog get() {
        while (itemCount == 0) {
            try {
                this.wait();
            } catch (InterruptedException e) {
            }
        }
        final Hotdog hotdog = buffer[front];
        front = (front + 1) % buffer.length;
        System.out.println(
            "Item count: " + itemCount + ", " +
            "Consuming " + hotdog
        );
        itemCount--;
        this.notifyAll();
        return hotdog;
    }
}

class Hotdog {

```

```java
    private int id;

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

    @Override
    public String toString() {
        return "Hotdog [id=" + id + "]";
    }
}

```

Compilation and execution

```
javac -d ../bin ProducerConsumer.java && java -cp ../bin ProducerConsumer 10 3
```

Output

```
Item count: 0, Producing Hotdog [id=0]
Item count: 1, Consuming Hotdog [id=0]
Item count: 0, Producing Hotdog [id=1]
Item count: 1, Producing Hotdog [id=2]
Item count: 2, Consuming Hotdog [id=1]
Item count: 1, Producing Hotdog [id=3]
Item count: 2, Producing Hotdog [id=4]
Item count: 3, Consuming Hotdog [id=2]
Item count: 2, Producing Hotdog [id=5]
Item count: 3, Consuming Hotdog [id=3]
Item count: 2, Producing Hotdog [id=6]
Item count: 3, Consuming Hotdog [id=4]
Item count: 2, Producing Hotdog [id=7]
Item count: 3, Consuming Hotdog [id=5]
Item count: 2, Producing Hotdog [id=8]
Item count: 3, Consuming Hotdog [id=6]
Item count: 2, Producing Hotdog [id=9]
Item count: 3, Consuming Hotdog [id=7]
Item count: 2, Consuming Hotdog [id=8]
Item count: 1, Consuming Hotdog [id=9]
```

Exercises

Factorial computation

- Write a single-threaded program which calculates `n!`, for each `n` running from `1` to `100,000`.
- Include in your program a timer to check a run time; consider using `System.currentTimeMillis()`).
- Design and implement a multi-threaded program to solve the same problem and show that runtime is indeed reduced.

Update shared array

- Write a program with two threads running concurrently, where both update a shared array.
- One thread puts numeric chars in the following order: `0`, `1`, ... , `9`.
- The other thread puts alphabetical characters in the following order: `A`, `B` ,..., `J`.
- **Bonus:** make sure the characters are inserted into the array in the alternating order, i.e.: `0`, `A`, `1`, `B`,...,`9`, `J`.