A Java Library for Concurrent Programming at Novice/Advanced Beginner Level

Jonas Mellin

October 28, 2017

1 Introduction

The purpose of this document is to describe the Java API for concurrent programming at novice/advanced beginner level. It is not intended for building real applications, since exception handling is avoided. That is, exceptions are only used for terminating the program when development faults occur, not general handling[1] of exceptions based on operational faults derived from the underlying system or other systems. Further, the interfaces are limited to enforce certain design principles as well as reduce the risk for misunderstandings. Finally, the API is meant to be used in situations where we employ formal methods tools to prove the correctness and avoiding the exceptions simplifies the proofs.

The Java API is inspired by the SR programming language, a pedagogical language and concurrent programming language for concurrent programming. In most cases, there are one-to-one mapping between constructs in SR and in the Java API.

2 An Example Program

The busy-wait solution for a two-process tiebreaker algorithm is found in figure 1 and in figure 2. First, this is an implementation of the interface Runnable in Java, which is used to specify the code of a thread. When a thread is created, an instance of a Runnable object can be passed as an argument. When the thread is started, the method *run()* is executed until termination. In this example, the processes does not terminate. At line 1, the specification "implements Runnable" means that objects of the class TwoProcessTieBreakerRunnable1 can be treated as a Runnable objects and are required to define the *run()* method. At line 4-5, the *run()* method is declared; the keyword *@Override* means that this overrides the default declaration in the Runnable interface and the *public void* ***run()*** means that the **run()** method is defined as open for access to all other objects and that it does not return any value.

[1] For example, recovering from an exception

In Java, there is no global variables. Instead, static member variables of a class are used. In all examples in this document, there is a class named **GlobalProgramState** that contains all global variables (i.e., accessible by all processes in a program) as public static member variables (as defined in lines 3-5 in figure 3). To access the global variables, prefix the variable with the class name **GlobalProgramState** as in line 7 and 8.

On line 10-12, the busy wait loop of the two process tie breaker are found. Process 1 continues to iterate as long as the global variable **in2** is true and **last** is equivalent to 1. On line 11, how to make a process sleep for a minimum delay is called.

The singleton class **GlobalProgramState** contains the global variables as well as the **main(String *argv*[])** code that is the entry point for starting the whole application. Essentially, the **main** method is employed to start all the processes. To make this process as convenient as possible as well as ensure that there is a close relationship to the SR examples, the creation and starting of processes are provided by non-standard methods in the **AndrewsProcess** class. On line 15-16, the start specification of the tie breaker processes are declared: one of each. On line 17, the processes are actually created and an array of **AndrewsProcess** objects are returned. This array can be passed to the method **startAndrewsProcesses** on line 18. If there are any kind of development faults that puts the program into an erroneous state, then this is caught by the catch clause on line 19 and the action is to print a stack trace (on line 20) to support localization of the fault

3 General Design Guidelines for the Concurrent Programming Course

1. Create a class named **GlobalProgramState** that contains the global variables as well as the **main** method. The **main** method should start all the processes.
2. Create a class that implements the **Runnable** interface for each kind of process.
3. Implement the **run()** method with the code. Do not used the constructor to initialize fields that depends on that the process actually exists. An instance of the **Runnable** is created for each process before the process is created and started. A simple solution is to make all initializations in the beginning of the **run()** method.

4 Shared Variable Mechanisms

The API employ standard Java API for shared variable mechanisms with recommendations for what methods that should be employed. Java, by default, support monitors with signal and continue policy on the signal and wait synchronization on condition variables. Condition variables are associated with the objects of a class, essentially, signal and wait are sent to the object containing the data instead of a separate condition variable.

In addition, in the package **java.lang.concurrent**, numerous synchronization and communication mechanisms are available. In the course, only semaphores will be employed directly and only a limited set of the semaphore functionality is allowed.

```java
public class TwoProcessTieBreakerRunnable1 implements Runnable {


  @Override
  public void run() {
    while (true) {
      GlobalProgramState.in1=true;
      GlobalProgramState.last=1;
      System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
   +" is waiting to access critical section");
      while (GlobalProgramState.in2 && GlobalProgramState.last==1) {
        AndrewsProcess.uninterruptibleMinimumDelay(10);
      }
      System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
   +" is in critical section");

      AndrewsProcess.uninterruptibleMinimumDelay(10);

      GlobalProgramState.in1=false;
      System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
   +" is not in critical section");
    }
  }


}
```

Figure 1: Main code of Tie Breaker process 1 in Java

```
1  public class TwoProcessTieBreakerRunnable2 implements Runnable {
2
3
4    @Override
5    public void run() {
6      while (true) {
7        GlobalProgramState.in2=true;
8        GlobalProgramState.last=2;
9        System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
       +" is waiting to access critical section");
10       while (GlobalProgramState.in1 && GlobalProgramState.last==2) {
11         AndrewsProcess.uninterruptibleMinimumDelay(10);
12
13       }
14       System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
       +" is in critical section");
15       GlobalProgramState.in2=false;
16       System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
       +" is not in critical section");
17     }
18   }
19
20
21 }
```

Figure 2: Main code of Tie Breaker process 2 in Java

```
1  public class GlobalProgramState {
2
3    static boolean in1=false;
4    static boolean in2=false;
5    static int last=1;
6
7
8    public static void main(String argv[]) {
9
10     System.out.print(AndrewsProcess.licenseText());
11
12     RunnableSpecification rs[]=new RunnableSpecification[2];
13     AndrewsProcess[] process;
14     try {
15       rs[0]=new RunnableSpecification(TwoProcessTieBreakerRunnable1.class
       ,1);
16       rs[1]=new RunnableSpecification(TwoProcessTieBreakerRunnable2.class
       ,1);
17       process = AndrewsProcess.andrewsProcessFactory(rs);
18       AndrewsProcess.startAndrewsProcesses(process);
19     } catch (InstantiationException | IllegalAccessException e) {
20       e.printStackTrace();
21     }
22   }
23
24 }
```

Figure 3: Main code of GlobalProgramState in TieBreaker for two processes

4.1 Semaphores

The semaphores in Java are defined in **java.lang.concurrent.semaphore** and it provide more functionality than P and V in the book. Essentially, there are numerous varieties of the methods that have a similar semantics as the P operation (e.g., **acquire**, **acquireUninterrupibly()**, **acquire(int permin)**, **tryAcquire()**). Out of these, the **acquireUninterrupibly()** is the method that have the same semantics as the P operation in SR and is the method that should be used in the course. The V operation is only represented by the **release()** method. In Java, *P(empty)* is the same as *empty.acquireUniterruptibly()* and *V(empty)* is the same as *empty.release()*.

In figure 4, the **GlobalProgramState** of the producer/consumer based on a single buffer is illustrated. There are three global variables: *buffer*, the buffer for communicating values of type int (in this example) between the producer and the consumer processes; *empty*, the semaphore that is used to synchronize producers so that they do not write unless there is space and only a single producer at the same time can be in the critical section; *full*, the semaphore that is used to synchronize consumers so that they only can consume when there is a value in the buffer. In Java, the initialization is done by allocation an object of the class **Semaphore** with the initial value of the semaphore as the initialization value.

On line 11 and 12, 10 producer processes and 20 consumer processes are cleclared. A **Producer** is a **Runnable** implementing an example of a producer semantics and **Consumer** is a **Runnable** implementing an example of a consumer semantics. Similarly to figure 3, the array of **RunnableSpecification** is passed to the factory resulting in an array of **AndrewsProcess** objects, which are then started.

The producer process (in figure 5), defined as a **Runnable**, in the class **Producer**, implements an example of a producer behavior. On line 8, the *empty* semaphore is acquired uninterrupibly[2]; if the $empty > 0$ the process acquires the semaphore and can pass, otherwise the process is blocked on the semaphore until another process releases the semaphore. In line 10, the buffer is assigned to a value and the local counter i is increased. After this, on line 11, the *full* semaphore is released.

Correspondingly, the consumer process (in figure 6), defined as a **Runnable** too, in the class **Consumer**, implements an example of consumer behavior. In this case, on line 7 the *full* semaphore is acquired, the value is consumed from the buffer on line 9, and on line 10 the *empty* semaphore is released.

4.2 Monitors

As mentioned, Java's default communication and synchronization mechanism is monitors. Each object of a class is a monitor if the class is defined in the correct way. The keyword in Java is **synchronized**, which can be used on method level as well as inside methods on member attributes. In this document, only the former is addressed since it most closely match the semantics of a monitor as expressed by Andrews [Andrews, 2000, ch. 5].

[2]That is, it is not terminated by an exception

```
1  public class GlobalProgramState {
2    public static int buffer;
3    public static Semaphore empty=new Semaphore(1);
4    public static Semaphore full=new Semaphore(0);
5
6    public static void main(String argv[]) {
7
8      System.out.print(AndrewsProcess.licenseText());
9
10     RunnableSpecification rs[]=new RunnableSpecification[2];
11     rs[0]=new RunnableSpecification(Producer.class,10);
12     rs[1]=new RunnableSpecification(Consumer.class,20);
13     try {
14       AndrewsProcess process[]=AndrewsProcess.andrewsProcessFactory(rs);
15       AndrewsProcess.startAndrewsProcesses(process);
16     } catch (InstantiationException e) {
17       e.printStackTrace();
18     } catch (IllegalAccessException e) {
19       e.printStackTrace();
20     }
21   }
22
23 }
```

Figure 4: GlobalProgramState of Producer/Consumer with a single buffer

The example used in section 4.1 based on producer/consumer is employed to illustrate monitors in Java. First, a class for defining the monitor behavior is required (see figure 7). There are two methods, *produce(int value)* and *consume()* that places a value in the buffer and consumes a value respectively. The *buffer* is defined on line 2 (cf. line 2 in figure 4), but, in contrast to lines 3-4 in figure **??** defining the semaphores, it is only necessary to have a boolean variables *full* on line 3 to check if the *buffer* is full or not.

On line 8-19, the *produce(int value)* is defined. Note the **synchronized** keyword which implies that the method is called in accordance to the signal and continue semantics. Essentially, it first checks if the buffer is full on line 9-15. The reason for the while-construct instead of an if statement is that this idiom is more robust. The reason is that it more robust to use *notifyAll()* rather than *notify()*, which leads to that all processes waiting in the monitor will be woken up. Each process test their condition and if, and only if, *full* is false will it exit the loop and continue. The signal and continue semantics implies that a process is in the monitor until it either exits the method or it calls *wait()* on an object. On line 16, the buffer is assigned a value, on line 17 the *full* flag is set to true and then *notifyAll()* is called on itself.

On line 21-33, the *consume()* method is defined. It is similar to the *produce(int value)* method. Note that the ordering on line 30-32 only works since the monitor follows a signal and continue semantics. If it would follow signal and wait, then the value of the buffer must be placed in a local process variable before calling *notifyAll()* and the return

```
1  public class Producer implements Runnable {
2
3
4    @Override
5    public void run() {
6      int i=1;
7      while(true) {
8        GlobalProgramState.empty.acquireUninterruptibly();
9        System.out.println("Process "+AndrewsProcess.currentAndrewsProcessId
       ()+" producing "+i);
10       GlobalProgramState.buffer=i++;
11       GlobalProgramState.full.release();
12     }
13   }
14
15 }
```

Figure 5: Producer example definition of Producer/Consumer with a single buffer

statement must return the copy of the buffer rather than the buffer itself, since the buffer value might have changed.

The declaration of a singleton object for the singleton monitor is found at line 2 in figure 8. This replaces line 2-5 in figure 4. The keyword **static** guarantees that one, and only one, object of this kind is assigned to the variable *buffer*. Apart from this, the example in figure 4 is similar to figure 8.

In figure 10, the producer is defined. This is less error-prone compared to the semaphore-based solution in figure 5. In figure **??**, the consumer is defined.

```
1  import se.his.iit.it325g.common.AndrewsProcess;
2
3  public class Consumer implements Runnable {
4
5
6    @Override
7    public void run() {
8      while(true) {
9        GlobalProgramState.full.acquireUninterruptibly();
10       int value=GlobalProgramState.buffer;
11       System.out.println("Process "+AndrewsProcess.currentAndrewsProcessId
     ()+": consuming value "+value);
12       GlobalProgramState.empty.release();
13
14     }
15   }
16
17 }
```

Figure 6: Consumer example definition of Producer/Consumer with a single buffer

5 Mechanisms based on Message Passing

5.1 Asynchronous Message Passing

5.2 Synchronous Message Passing

5.3 Rendezvous

6 Application Programming Interfaces

6.1 AndrewsProcess class

6.2 Chan class

7 How To Guide

7.1 Define a Process

7.2 Start a Process

7.3 Keep Track of Data Concerning a Process

7.4 Synchronize Processes

7.5 Communicate between Processes

7.6 Initialize a Program

7.7 Terminate a Program

References

[Andrews, 2000] Andrews, G. R. (2000). *Foundations of multithreaded, parallel, and distributed programming*. Addison-Wesley, Reading, Mass.

```
1  public class SingleBufferMonitor {
2    private int buffer;
3    private boolean full=false;
4
5    public SingleBufferMonitor() {
6    }
7
8    public synchronized void produce(int value) {
9      while(full) {
10       try {
11         this.wait();
12       } catch (InterruptedException ie) {
13         // do nothing
14       }
15     }
16     this.buffer=value;
17     this.full=true;
18     this.notifyAll();
19   }
20
21   public synchronized int consume() {
22     while (!full) {
23       try {
24         this.wait();
25       } catch (InterruptedException ie) {
26         // do nothing
27       }
28
29     }
30     this.full=false;
31     this.notifyAll();
32     return this.buffer;
33   }
34
35 }
```

Figure 7: SingleBufferMonitor of Producer/Consumer with a single buffer, based on monitors

```
public class GlobalProgramState {
  public static SingleBufferMonitor buffer=new SingleBufferMonitor();

  public static void main(String argv[]) {

    System.out.print(AndrewsProcess.licenseText());

    RunnableSpecification rs[]=new RunnableSpecification[2];
    rs[0]=new RunnableSpecification(Producer.class,10);
    rs[1]=new RunnableSpecification(Consumer.class,20);
    try {
      AndrewsProcess process[]=AndrewsProcess.andrewsProcessFactory(rs);
      AndrewsProcess.startAndrewsProcesses(process);
    } catch (InstantiationException e) {
      e.printStackTrace();
    } catch (IllegalAccessException e) {
      e.printStackTrace();
    }
  }

}
```

Figure 8: GlobalProgramState of Producer/Consumer with a single buffer, based on monitors

```
public class Producer implements Runnable {


  @Override
  public void run() {
    int i=1;
    while(true) {
      System.out.println("Process "+AndrewsProcess.currentAndrewsProcessId
    ()+" producing "+i);
      GlobalProgramState.buffer.produce(i++);
    }
  }

}
```

Figure 9: Producer of Producer/Consumer with a single buffer, based on monitors

```
public class Consumer implements Runnable {


  @Override
  public void run() {
    while(true) {
      int value=GlobalProgramState.buffer.consume();
      System.out.println("Process "+AndrewsProcess.currentAndrewsProcessId
    ()+": consuming value "+value);

    }
  }

}
```

Figure 10: Consumer of Producer/Consumer with a single buffer, based on monitors