Concurrency Exercises 3: Monitors

Topic 4.1: Concurrency Patterns: Monitors

Begin by reading the original papers on monitors that are recommended in the lectures.

The following questions are courtesy of Magee & Kramer, chapter 5.

1. A single-slot buffer may be modelled by:

   \[
   \text{ONEBUF} = (\text{put} \rightarrow \text{get} \rightarrow \text{ONEBUF}).
   \]

   Write a Java class, \texttt{OneBuf}, that implements this one-slot buffer as a monitor.

   Answer:

   ```java
   /* java Implementation
   public class OneBuf {
   Object slot = null;

   public synchronized void put(Object o) throws InterruptedException {
   while(slot != null) wait();
   slot = o;
   notifyAll();
   }

   public synchronized Object get () throws InterruptedException {
   while(slot == null) wait();
   Object o = slot;
   slot = null;
   notifyAll();
   return o;
   }
   }
   ```

2. In the museum example on Exercise Sheet 2, identify which of the processes, \texttt{EAST}, \texttt{WEST}, \texttt{CONTROL}, and \texttt{DIRECTOR}, should be threads and which should be monitors. Provide an implementation of the monitors.

   Answer:

   EAST, WEST, DIRECTOR will be implemented as threads, CONTROL as a monitor.
CONTROL = CLOSED[0],
CLOSED[i:0..N] = (when (i==0) open -> OPENED[0]
 |when (i>0) leave -> CLOSED[i-1]
 ),
OPENED[i:0..N] = (close -> CLOSED[i]
 |when (i<N) arrive -> OPENED[i+1]
 |when (i>0) leave -> OPENED[i-1]
 ).

/* java Implementation

public class Control {
    int count == 0;
    boolean opened = false;

    public synchronized void arrive() throws InterruptedException {
        while(!opened) wait();
        ++count
    }

    public synchronized void leave() throws InterruptedException {
        while(count<=0) wait();
        --count;
        if (count==0) notifyAll();
    }

    public synchronized void open() throws InterruptedException {
        while (count>0) wait();
        opened =true;
        notifyAll();
    }

    public synchronized void close() throws InterruptedException {
        opened = false;
    }
}

3. The Dining Savages: A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty, in which case he waits for the pot to be filled. If the pot is empty, the cook refills the pot with M servings. The behaviour of the savages and cook are described by:

    SAVAGE = (getserving -> SAVAGE).
    COOK = (fillpot -> COOK).

Model the behaviour of the pot as an FSP process and then implement it as a Java monitor.
Answer:
const M = 5

SAVAGE = (getserving -> SAVAGE).

COOK = (fillpot -> COOK).

POT = SERVINGS[0],
SERVINGS[i:0..M] = (when (i==0) fillpot -> SERVINGS[M]
|when (i>0 ) getserving -> SERVINGS[i-1])

||SAVAGES = (SAVAGE || COOK || POT).