Producer and Consumer, Synchronization
- Abhishek Yeluri and Yashwanth Reddy
Assignment Review

• You are supposed to implement a solution to Producer and Consumer problem, using a circular FIFO buffer.
• There can be one or more than one producers and consumers.
• The producers are supposed to generate a certain number of elements. Elements will be randomly generated characters.
• Consumers are supposed to consume the elements, produced by Producers.
• Both producers and consumers are supposed to report the character produced/consumed along with the index and timestamp with nanosecond resolution.
Producer.java

- The number of producers will be given the 3rd argument. It will be a proper factor of the number of elements to be produced.
- Each producer will produce the (total number of elements / number of producers) elements.
- The seed is used to set the random number generator to generate the same sequence every time the same seed is given.
- Generate a number between 1 and 26 (both inclusive) and map this on to an alphabet.
- A producer cannot insert an element into the buffer when the buffer is full.
Consumer.java

- A consumer consumes an element from the buffer.
- Each consumer will consume the \((\text{total number of elements} \div \text{number of consumers})\) elements.
- A consumer cannot consume an element when the buffer is empty.
Buffer.java

- Buffer.java contains the circular FIFO buffer that will be used among all the producers and the consumers.
- It also has the required functions that is used to insert or remove an element, and it returns the appropriate values.
- It may additionally have, other functions such as isFull(), isEmpty(), etc. depending on your implementation.
ProducerConsumer.java

- This is acts as the Starter program for this assignment.
- It creates one instance of the buffer, and creates required number of threads of producers and consumers, and then waits for all of them to finish.
Synchronization in Java

- Java has inbuilt monitors
  - Allows threads to have mutual exclusion
  - Allows threads the ability to wait (block) for a condition to become true
- Built in thread class can be extended and used
  - Instantiate and use myThread.start()
  - @Override run() to change what a thread does
- Signalling is done using
  - wait()
  - notify() or notifyAll()
public class PhilosopherThread extends Thread
{
    @Override
    public void run()
    {
        //The entry point for each thread
    }
}
Creating and Starting threads

PhilosopherThread Socrates = new PhilosopherThread(table, seat);
Socrates.start(); //begins our Socrates thread and invokes the run() method
Synchronization
Synchronized methods

- A piece of logic marked with synchronized becomes a synchronized block, allowing only one thread to execute at any given time.

```java
public synchronized void pickup(int i) throws InterruptedException {
    //Synchronized code goes in here
}
```
wait(), notify() and notifyAll()

- **wait()**
  - Causes current thread to wait until another thread invokes the notify() or notifyAll() method

- **notify()**
  - notify() wakes up one thread waiting for the lock

- **notifyAll()**
  - The notifyAll() method wakes up all the threads waiting for the lock; the JVM selects one of the threads from the list of threads waiting for the lock and wakes that thread up
Makefile

COMPILER= javac
JRE= java
FILES= DiningPhilosophers.java PhilosopherTable.java PhilosopherThread.java
EXE= DiningPhilosophers
all:
    $(COMPILER) $(FILES)
run:
    $(JRE) $(EXE)
clean:
    rm *\.class
JFLAGS = -g
JC = javac
.SUFFIXES: .java .class
.java.class: $(JC) $(JFLAGS) $*.java
CLASSES = \ DiningPhilosophers.java
\ \ PhilosopherTable.java
\ \ PhilosopherThread.java
default: classes
classes: $(CLASSES:.java=.class)
clean: $(RM) *.class