CS240 Operating Systems, Communications and Concurrency

Classical Coordination Problems

Semaphores are a general purpose synchronisation tool that can be used to solve a variety of synchronisation problems.

We will look next at three classical coordination problems to which they can be applied.

We are going to simulate Semaphores and their indivisible P and V operations by defining a Java Class

CS240 Operating Systems, Communications and Concurrency

Java Synchronisation and Intrinsic Object Locks

Every Java object has an intrinsic lock, managed by an inbuilt monitor. Ordinarily this lock is ignored when the object is being referenced through its methods.

When a method within the object is declared to be **synchronized** however, the calling thread requires ownership of the object's lock before the method can be executed.

If the lock is in use, the calling thread blocks and is placed on an entry queue (or set) for the object's lock. If the lock is free, the calling thread becomes the owner. The lock is released when the calling thread exits the object's method.

CS240 Operating Systems, Communications and Concurrency

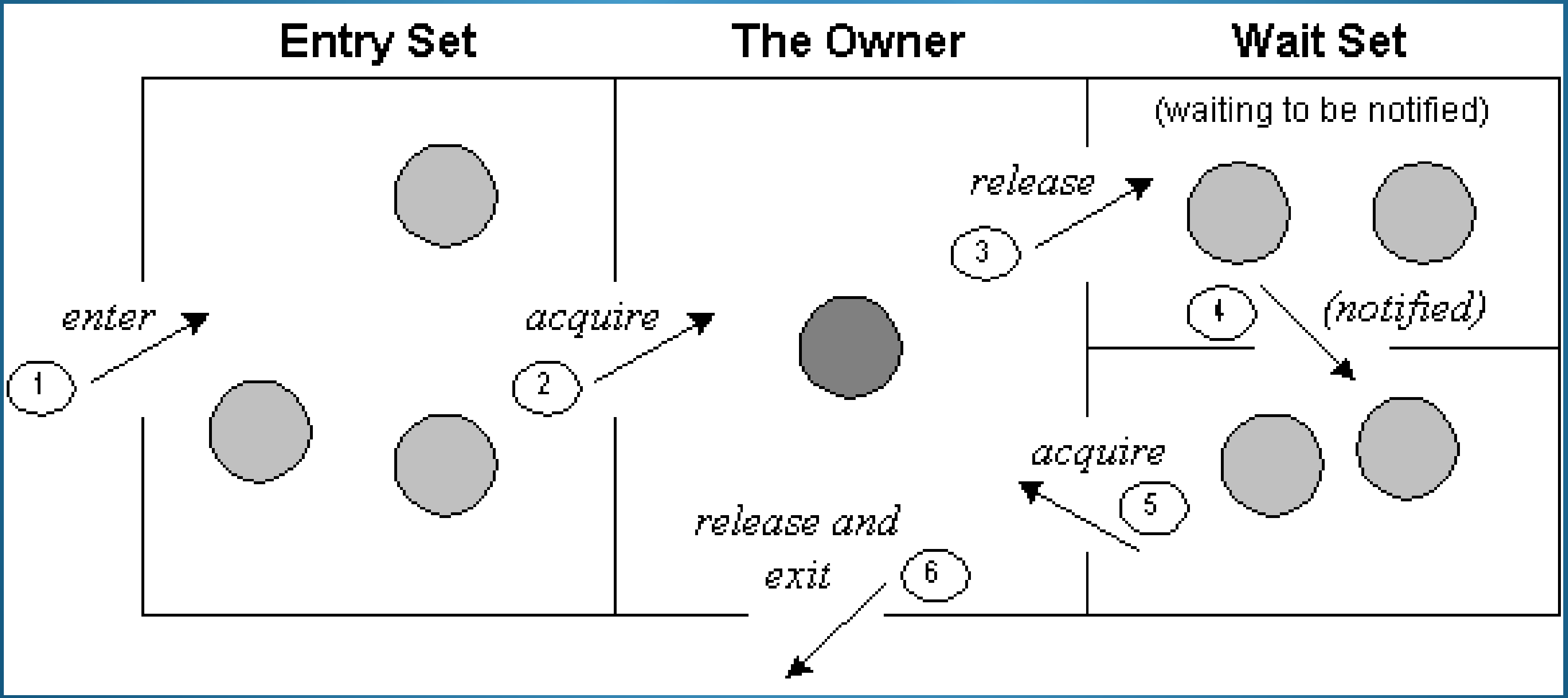
Wait and Notify

Every object in Java has associated with it a wait set which is initially empty. When a thread which holds the object's lock calls the **`wait()`** method the following happens:-

The thread releases the associated object lock
The thread is set to blocked
The thread is placed in the wait set for that object.

The **notify()** method picks an arbitrary thread from the wait set so it can compete to reacquire the lock. The thread is made runnable. When it eventually reacquire's the object lock, it returns from the original wait method call it made.

CS240 Operating Systems, Communications and Concurrency



CS240 Operating Systems, Communications and Concurrency

```
class Semaphore {
    private int value;


// Constructor
    public Semaphore(int value) {
        this.value = value;
    }
```

CS240 Operating Systems, Communications and Concurrency

```
public synchronized void acquire() {
      while (value == 0) {
         try {
// Calling thread waits until semaphore is free
              wait();
         } catch(InterruptedException e) {}
      }
   value = value - 1;
   }

public synchronized void release() {
   value = value + 1;
   notify();
}
```

CS240 Operating Systems, Communications and Concurrency

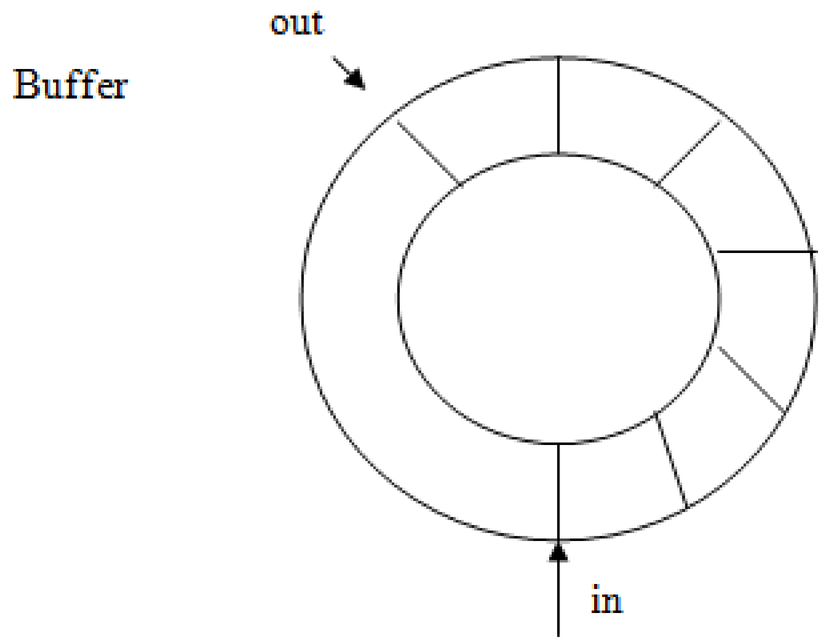
The Producer/Consumer or Bounded Buffer Problem

We have a system of processes where some produce items of output while others consume (or continue to process) these items. As the production and consumption is done at different times and rates (asynchronously) the processes use a shared buffer.

Any staged software system or communication between cooperating processes might behave like this.

For example, the Unix system uses the pipe mechanism to facilitate transfer of output from one command to be used as input by another. A compiler generates results in stages from lexical analysis to semantic analysis and then to code generation.

CS240 Operating Systems, Communications and Concurrency



<u>Producer Process Behaviour</u>

```
while (true) {
   Produce Object item
   Insert item in buffer
}
```

<u>Consumer Process Behaviour</u>

```
while (true) {
   Remove item from buffer
   Consume item
}
```

Approach to Synchronisation (Use 3 Semaphores)

A producer process can only put an item into the buffer if a free space exists. We will use a semaphore called **empty** to count empty spaces and hold up a producer if the buffer has no empty space and to signal when a space is created.

A consumer can only take an item from the buffer if it is not empty. We will use another signalling semaphore called **full** to count items and hold up a consumer if the buffer is empty.

Modification of the buffer involves changing the values of the in and out variables and the contents of the buffer array. These modifications must be serialised. We will use a semaphore **mutex** for controlling buffer modifications.

CS240 Operating Systems, Communications and Concurrency

```
public class Buffer {

    private static final int BUFFER_SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;
```

CS240 Operating Systems, Communications and Concurrency

```
public class Buffer {


// The constructor for Buffer
 public Buffer() {
      in = 0;
      out = 0;
      buffer = new Object[BUFFER_SIZE];

      mutex = new Semaphore(1);
      empty = new Semaphore(BUFFER_SIZE);
      full = new Semaphore(0);
   }
```

Binary semaphore

Counting semaphores

CS240 Operating Systems, Communications and Concurrency

```
public class Buffer {

public void insert(Object item) {
      empty.acquire();

      mutex.acquire();
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      mutex.release();

      full.release();
   }
```

CS240 Operating Systems, Communications and Concurrency

```
public class Buffer {

public Object remove() {
       full.acquire();

       mutex.acquire();
       Object item = buffer[out];
       out = (out + 1) % BUFFER_SIZE;
       mutex.release();

       empty.release();
       return item;
   }
```

Defining the producer thread's behaviour

Next we define classes for creating runnable objects with the desired producer or consumer behaviour, that is, code that can be executed by a thread. Such classes must define a `run()` method.

A thread given a runnable object will execute its `run()` method.

The main program should create a buffer object first to be shared between producer and consumer threads and then pass a reference to this buffer to the runnable object's constructor as shown next.

Defining the producer thread's behaviour

```
public class Producer implements Runnable {
   private Buffer buffer;

   public Producer(Buffer buffer) {
      this.buffer = buffer;
   }
```

Pass in the shared buffer to the constructor

Defining the producer thread's behaviour

```
import java.util.Date;
public class Producer implements Runnable {

public void run() {
    Date message;

    while (true) {
        message = new Date();
        buffer.insert(message);
        System.out.println("Inserted "+ message);
    }
}
```

Defining the producer thread's behaviour

```
import java.util.Date;
public class Producer implements Runnable {

public void run() {
   Date message;

   while (true) {
      message = new Date();
      try {
         Thread.sleep(2000);
      } catch (InterruptedException e) {}
      buffer.insert(message);
      System.out.println("Inserted "+ message);
   }
}
```

Slow down the loop a bit (by 2 seconds)

Defining the consumer thread's behaviour

```
public class Consumer implements Runnable {
   private Buffer buffer;

   public Consumer(Buffer buffer) {
      this.buffer = buffer;
   }
```

Pass in the shared buffer to the constructor

Defining the producer thread's behaviour

```
import java.util.Date;
public class Consumer implements Runnable {

public void run() {
      Date message;

      while (true) {
         try {
            Thread.sleep(1000); // sleep for 1000 ms
         } catch (InterruptedException e) {}
         message = (Date) buffer.remove();
         // consume the item
         System.out.println("Removed "+ message);
      }
   }
```

The main routine of the Bounded Buffer Simulation

The code creates a Buffer object to be shared between the producers and consumers.

It then creates creates two threads. It passes a runnable object to the constructor of each thread.

The first gets an instantiation of the `Producer` class and the second gets an instantiation of the `Consumer` class.

Invoking the `start()` method causes each thread to execute the `run()` method of its runnable object.

CS240 Operating Systems, Communications and Concurrency

```
public class BoundedBufferSimulation {
    public static void main (String args[]) {
        Buffer buffer = new Buffer();

        // Create one producer and one consumer process
        Thread producer1 = new Thread(new Producer(buffer));
        Thread consumer1 = new Thread(new Consumer(buffer));

        producer1.start();
        consumer1.start();
    }
}
```