CS240 Operating Systems, Communications and Concurrency – Dermot Kelly
Practical 7 – Thread Programming in Java and C
The Producer Consumer Bounded Buffer Synchronisation Problem

The Producer-Consumer problem is an example of a concurrency problem dealing with the management of a shared finite buffer structure used for exchanging data between processes which are running asynchronously. Some processes write data into the buffer (producers) and other processes remove data from the buffer (consumers). The approach to coding of a solution to the producer-consumer problem was covered in lectures and the corresponding Java code for a simulation of this problem is given below.

First create a new folder in your CYGWIN home/cs240 directory named p7.
The solution below uses semaphore operations provided by a `Semaphore` class with synchronised methods `acquire()` and `release()` which operate on a local integer `value` belonging to an instance of `Semaphore`. A thread trying to acquire the semaphore behaves as follows: - If the value of the semaphore is greater than 0, the calling thread decrements it by 1, otherwise the calling thread is blocked by invoking `wait()`. When a thread invokes `release()`, it increments the semaphore and notifies another thread on the queue that the semaphore may be available. That thread will then wake up and continue its attempt to acquire it. See if you can follow the description of these methods in the code below. **Save the Semaphore class below in a file called "Semaphore.java"**.

```
/* The Semaphore class contains methods declared as
synchronized. Java's locking mechanism will ensure
that access to Semaphore methods is mutually exclusive
among threads that invoke these methods.
*/
class Semaphore {
	private int value;

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

	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();
	}
}
```

Recall from lectures - Wait and Notify
Every object in Java has associated with it a wait set which is initially empty. When a thread calls the `wait()` method the following happens:- The thread releases the object lock; The thread is set to blocked; The thread is placed in the wait set for the object.
The `notify()` method picks an arbitrary thread from the wait set and moves it to the entry set where is can reacquire the lock. The thread is made runnable.

The following class attempts to coordinate the actions of producer and consumer processes when accessing the buffer. A producer can only write into the buffer when a space exists for its item, and a consumer can only remove an item from the buffer when one exists. At all times, there should only be at most one process actually manipulating the buffer structure itself. Three semaphores are used to control these three conditions, `full`, `empty` and `mutex` as described in the lecture.

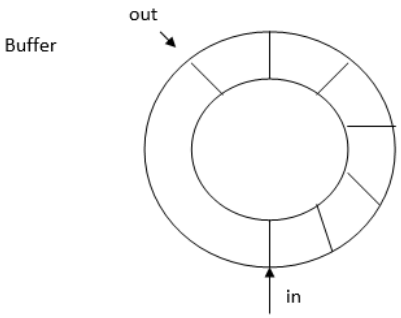
An example implementation of the buffer class, **save as file "Buffer.java"** :-

```
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;

	public Buffer() {
		in = 0;
		out = 0;
		buffer = new Object[BUFFER_SIZE];
		mutex = new Semaphore(1);
		empty = new Semaphore(BUFFER_SIZE);
		full = new Semaphore(0);
	}

	public void insert(Object item) {
		empty.acquire();
		mutex.acquire();
		   buffer[in] = item;
		   in = (in + 1) % BUFFER_SIZE;
		mutex.release();
		full.release();
	}

	public Object remove() {
		full.acquire();
		mutex.acquire();
		   Object item = buffer[out];
		   out = (out + 1) % BUFFER_SIZE;
		mutex.release();
		empty.release();
		return item;
	}
}
```



Producer Process Behaviour

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

Consumer Process Behaviour

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

An example behaviour for a producer thread is given below.

This class defines a runnable object, that is, an object that can be executed by a thread. The thread executing this object will execute its `run()` method where the producer behaviour is described below. 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 below. **Save this class as "Producer.java".**

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

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

   public void run() {
      Date message;

      while (true) {
         message = new Date(); // produce an item
         try {
            Thread.sleep(1000); // sleep for 1000 ms
         } catch (InterruptedException e) {}
         buffer.insert(message);
         System.out.println("Inserted "+ message);
      }
   }
}
```

An example behaviour for a consumer thread is given below, **save as "Consumer.java"**

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

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

   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, **save as "BoundedBufferSimulation.java"**
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.

```
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();
   }
}
```

Compile all the five code files using `javac *.java` **and run the BoundedBufferSimulation in a JVM with the command** `java BoundedBufferSimulation`
Do not continue until this program is working.

<u>Modify the program as follows:</u>
Copy the Producer.java, Consumer.java and BoundedBufferSimulation.java files to **Producer2.java**, **Consumer2.java** and **BoundedBufferSimulation2.java**, you can make modifications to these without affecting the original files.

Change the names of the **classes** and their **constructors** within those 3 new files to Producer2, Consumer2 and BoundedBufferSimulation2. In the BoundedBufferSimulation2 `main()` method modify the name of the constructors to Producer2 and Consumer2.

```
      Thread producer1 = new Thread(new Producer2(buffer));
      Thread consumer1 = new Thread(new Consumer2(buffer));
```

Just to make sure you did all this properly, compile all the java code with `javac *.java` and run the BoundedBufferSimulation2 with `java BoundedBufferSimulation2`
It should work the same as the original.

The modifications you make below should only be done to these three new files, not the originals used earlier.

The constructors for the Producer2 and Consumer2 classes currently take a single parameter (a reference to the shared buffer object) which is stored in a local variable of the same name within the object. Modify the constructors to take an additional integer parameter representing an integer id, and store it in a corresponding instance variable within the object when it is created. Modify the Producer2 and Consumer2 run methods by changing the println statement in their loops so that the integer id given to the constructor can be printed each time the corresponding thread executes its run loop, and inserts or removes an item, so we can know which Producer/Consumer is generating the message.

Modify the main program code in BounderBufferSimulation2.java so that it creates two consumer and two producer threads. We need to pass two parameters now to the new constructors here, not just buffer, but also an integer representing a different ID for each producer, e.g. 1 and 2, and each consumer). Don't forget to start all 4 threads. All 4 threads should then execute concurrently using the same shared buffer.

The code below is an alternative multithreaded implementation of the producer/consumer bounded buffer synchronisation problem written in C and using the POSIX pthreads API.
From your understanding of the Java implementation, you should easily be able to follow the code and understand the synchronisation. Save the file below as "boundedbuffer.c", and compile using `cc boundedbuffer.c -o boundedbuffer` or if you need to link the pthread library:-
`cc boundedbuffer.c -l pthread -o boundedbuffer`

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <semaphore.h>

#define BUFFER_SIZE 10
#define TRUE 1

time_t buffer[BUFFER_SIZE];
int in,out;
pthread_mutex_t mutex;
sem_t empty, full;

void init_buffer()
   {
       in = 0;
       out = 0;
       pthread_mutex_init(&mutex,NULL); /* serialise buffer modifications */
       /* 2nd parameter indicates semaphore shared between threads not processes */
       sem_init(&empty,0,BUFFER_SIZE); /* buffer has max free space */
       sem_init(&full,0,0); /* No items in buffer */
   }


void insert_item(time_t timeinseconds)
   {
       sem_wait(&empty);
       pthread_mutex_lock(&mutex);
              buffer[in] = timeinseconds;
              in = (in + 1) % BUFFER_SIZE;
       pthread_mutex_unlock(&mutex);
       sem_post(&full);
   }


time_t remove_item()
   {
      time_t item; /* item stores date as a number of seconds since Jan 1 1970. */
       sem_wait(&full);
       pthread_mutex_lock(&mutex);
              item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
       pthread_mutex_unlock(&mutex);
       sem_post(&empty);
    return item;
   }
```

```
void *consumer(void* id)
  {
       unsigned long my_id = (unsigned long) id;
       time_t timeinseconds;

       while (TRUE) {
              timeinseconds = remove_item(); /* Consume the date item */
              printf("Consumer %ld Removed %ld \n",my_id, timeinseconds);
           sleep(5); // the consumer operates more slowly than producer
       }
}

void *producer(void* id)
   {
       unsigned long my_id = (unsigned long) id;
       time_t timeinseconds;

       while (TRUE) {
              timeinseconds = time(NULL);  /* Produce a date item */
              insert_item(timeinseconds);
              printf("Producer %ld Inserted %ld \n",my_id, timeinseconds);
              sleep(1); // the producer operates more quickly than consumer
       }
}


int main(int argc, char **argv) {
       pthread_t thread1, thread2, thread3, thread4;
       unsigned long p1 = 1;
       unsigned long c1 = 1;
       unsigned long p2 = 2;
    unsigned long c2 = 2;
       init_buffer();

       /* Start 2 producer and 2 consumer threads and give them ids*/
       pthread_create(&thread1, NULL, consumer, (void *) p1);
       pthread_create(&thread2, NULL, producer, (void *) c1);
       pthread_create(&thread3, NULL, consumer, (void *) p2);
       pthread_create(&thread4, NULL, producer, (void *) c2);

       /* Execute pthread_join to prevent main thread from exiting and terminating program */
       pthread_join(thread1, NULL);
       pthread_join(thread2, NULL);
       pthread_join(thread3, NULL);
       pthread_join(thread4, NULL);
       printf("All threads finished\n"); /* Will never happen as threads loop indefinitely */
       exit(0);
}
```

When the program runs, notice the rate at which output messages from the producer and consumer threads appear initially and then notice the point where this rate slows down. Can you explain this from the code?

The producer and consumer loops take different amounts of time due to the different sleep parameters used. When the buffer is empty at the start, the producers operate at full speed but when the buffer becomes full, the producers are limited to operating at the speed at which the consumer removes items, so you can see the synchronisation is working from the behaviour of the program.

What to Submit on Moodle

A single screenshot containining the following:-

Open 4 terminals, and set current directory in each to your practical 7 directory “cd cs240/p7” .
Run each of the commands below in one of the terminal windows.

Type `ls -l` (missing a long directory listing will cost you marks)
Type `java BoundedBufferSimulation`
Type `java Bounded BufferSimulation2`
Type `./boundedbuffer`

Take a screenshot, submit it on moodle.