COM1032 Mobile Computing
Lab 4
Introduction to Threads

Purpose:

The purpose of this lab session is to familiarise yourself with Multithreading and identify why threads are different from processes and how they are handled differently by the Operating Systems.

Aim

By the end of the lab you will be able to:

- Understand the multithreads programs APIs, execution and scheduling requirements in different operating systems.
- Able to create multithreaded programs using Java in various OS Systems.

Posix Threads

`Pthreads` refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a specification for thread behaviour, not an implementation. Operating system designers may implement the specification in any way they wish. Numerous systems implement the `Pthreads` specification, including Solaris, Linux, Mac OS X, and Tru64 UNIX. Shareware implementations are available in the public domain for the various Windows operating systems as well.

Download `pthreads UnixExamp.c` from SurreyLearn to compare the syntax with the Java Threads that we will focus on in this module. The C code demonstrates the basic `pthreads` API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread. In a `pthreads` program, separate threads begin execution in a specified function. In Figure 4.9, this is the `runner()` function. When this program begins, a single thread of control begins in `main()`. After some initialization, `main()` creates a second thread that begins control in the `runner()` function. Both threads share the global data sum. All `pthreads` programs must include the `pthread.h` header file. The statement `pthread_t tid` declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The `pthread attr_t attr` declaration represents the attributes for the thread. We set the attributes in the function call `pthread attr init(&attr).` Because we did not explicitly set any attributes, we use the default attributes provided.

A separate thread is created with the `pthread create()` function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution—in this case, the `runner()` function. Last, we pass the integer parameter that was provided on the command line, `argv[1]`.

At this point, the program has two threads: the initial (or parent) thread in `main()` and the summation (or child) thread performing the summation operation in the `runner()` function. After creating the summation thread, the parent thread will wait for it to complete by calling the `pthread join()` function. The summation thread will complete when it calls the function `pthread exit()`. Once the summation thread has returned, the parent thread will output the value of the shared data sum. You can run and execute this code in Linux and Raspberry Pi using the following commands:

```
gcc -o pthreadsUnixExamp pthreadsUnixExamp.c
./pthreadsUnixExamp 100
```

Win32 Threads

The Win32 thread API is Windows `pthreads` implementation. It is illustrated in the C program `pthreadsWinExamp.c` that you can download from SurreyLearn. Notice that we must include the `windows.h` header file when using the Win32 API. Just as in the `pthreads` version shown in Figure 4.9, data shared by the separate threads—in this case, Sum—are declared globally (the DWORD data type is an unsigned 32-bit integer). We also define the `Summation()` function that is to be performed in a separate thread. This function is passed a pointer to a void, which Win32 defines as `LPVOID`. The thread performing this function sets the global data Sum to the value of the summation from 0 to the parameter passed to `Summation()`.

Threads are created in the Win32 API using the `CreateThread()` function, and—just as in `pthreads`—a set of attributes for the thread is passed to this function. These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state. In this program, we use the default values for these attributes (which do not initially set the thread to a suspended state and instead make it eligible to be run by the CPU scheduler). Once the summation thread is created, the parent must wait for it to complete before outputting the value of Sum, since the value is set by the summation thread. Recall that the `pthread` program (`pthreadsUnixExamp.c`) has the parent thread wait for the summation thread using the `pthread join()` statement. In `pthreadsWinExamp.c`, the equivalent of this in the Win32 API is performed using the `WaitForSingleObject()` function, which causes the creating thread to block until the summation thread has exited.

Java Thread

There are two techniques for creating threads in a Java program. One approach is to create a new class that is derived from the `Thread` class and to override its `run()` method. However, the most common technique is to define a class that implements the `Runnable` interface. The `Runnable` interface is defined as follows:

```
public interface Runnable {
         public abstract void run();
}
```

When a class implements `Runnable`, it must define a `run()` method. The code implementing the `run()` method is what runs as a separate thread.

File "`Summation.java`" shows the Java version of a multithreaded program that determines the summation of a non-negative integer. The `Summation` class implements the `Runnable` interface. Thread creation is performed by creating an object instance of the `Thread` class and passing to the constructor a `Runnable` object.

Creating a Thread object does not specifically create the new thread; rather, it is the `start()` method that actually creates the new thread. Calling the `start()` method for the new object does two things:

1. It allocates memory and initializes a new thread in the JVM.
2. It calls the `run()` method, making the thread eligible to be run by the JVM. ***(Note that we never call the `run()` method directly. Rather, we call the `start()` method, and it calls the `run()` method on our behalf.)***

Ex1: a) Will the following code snippets create threads that run simultaneously, or run sequentially in the main thread?

```
public class Test {
         public static void main(String[] args) {
                  Thread thrd1 = new Thread(new Runnable() {
                           public void run() {
                                    for (int i=0;i<100;i++)
                                             System.out.println(i + ": Inside Thread 1");
                           }
                  });

                  Thread thrd2 = new Thread(new Runnable() {
                           public void run() {
                                    for (int i=0;i<100;i++)
                                             System.out.println(i + ": Inside Thread 2");
                           }
                  });
                  thrd1.run();
                  thrd2.run();
         }
}
```

b) How do you fix this code such that both threads run simultaneously? Do you see the order of the printing still sequential after fixing the code? Can you explain the order of printing when you run several times?

When the summation program runs, two threads are created by the JVM. The first is the parent thread, which starts execution in the `main()` method. The second thread is created when the `start()` method on the Thread object is invoked. This child thread begins execution in the `run()` method of the Summation class. After outputting the value of the summation, this thread terminates when it exits from its `run()` method.

Sharing of data between threads occurs easily in Win32 and pthreads, as shared data are simply declared globally. As a pure object-oriented language, Java has no such notion of global data; if two or more threads are to share data in a Java program, the sharing occurs by passing references to the shared object to the appropriate threads. In the Java program "`Summation.java`", the main thread and the summation thread share the object instance of the Sum class. This shared object is referenced through the appropriate `getSum()` and `setSum()` methods. (You might wonder why we don't use a `java.lang.Integer` object rather than designing a new Sum class. The reason is that the `java.lang.Integer` class is immutable—that is, once its integer value is set, it cannot change.)

Recall that the parent threads in the pthreads and Win32 libraries use `pthread_join()` and `WaitForSingleObject()` (respectively) to wait for the summation threads to finish before proceeding.

The `join()` method in Java provides similar functionality. Notice that `join()` can throw an `InterruptedException`, which we choose to ignore for now.

Java actually identifies two different types of threads: (1) daemon (pronounced "demon") and (2) non-daemon threads. The fundamental difference between the two types is the simple rule that the JVM shuts down when all non-daemon threads have exited. Otherwise, the two thread types are identical. When the JVM starts up, it creates several internal daemon threads for performing tasks such as garbage collection. A daemon thread is created by invoking the `setDaemon()` method of the Thread class and passing the method the value true. For example, we could have set the thread in the "`Summation.java`" program as a daemon by adding the following line after creating —but before starting—the thread:

```
thrd.setDaemon(true);
```

Java Thread States

A Java thread may be in one of six possible states in the JVM as shown in Figure 1.

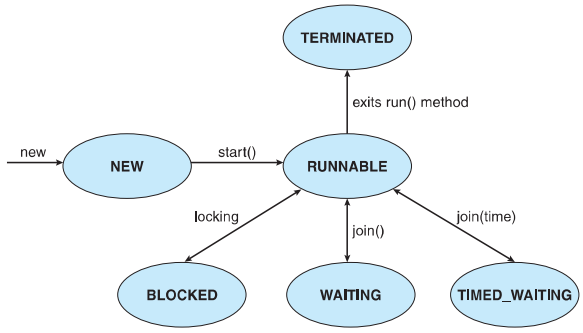

Figure 1: Java Thread States

The Java API for the Thread class provides several methods to determine the state of a thread. The `isAlive()` method returns true if a thread has been started but has not yet reached the Terminated state; otherwise, it returns false. The `getState()` method returns the state of a thread as an enumerated data type as one of the values shown in the documentation and corresponding to the states in Figure 1:

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Thread.State.html

A Multithreaded Solution to the Producer–Consumer Problem

Last week, you implemented a producer/consumer solutions using producer/consumer objects created in the main thread, and both using a shared buffer as shared memory to access.

Ex 2: Can you discuss some of the problems of the solution of last week?

Now that you know how to create threads, we will recreate the producer/consumer problem using Java Threads and solve some of the problems you encountered:

From SurreyLearn, download the following Java files:

`Channel.Java:` Implements an interface for message passing, with methods send and receive.

`MessageQueue.Java:` It implements the Channel interface. The buffer is implemented using the `java.util.Vector` class, meaning that it will be a buffer of unbounded capacity. Also note that both the `send()` and `receive()` methods are nonblocking.

`SleepUtilities.Java:` A class containing two overloaded nap methods for sleeping the current calling thread, one uses a constant sleep duration, and one uses a passed argument.

`Producer.Java:` The producer class overrides the run method it inherited from the `Runnable` interface it is implementing. In the run method, it alternates among sleeping for a while, producing an item, and entering that item into the queue. The sleeping is implemented in the `SleepUtilities` class.

`Consumer.Java:` The consumer class also overrides the run method it inherited from the Runnable interface it is implementing. In the run method, it alternates between sleeping and then retrieving an item from the queue and consuming it. Because the `receive()` method of the `MessageQueue` class is non-blocking, the consumer must check to see whether the message that it retrieved is null.

`Factory.Java:` The Factory class creates a message queue for buffering messages, using the `MessageQueue` class. It then creates separate producer and consumer threads, and passes each thread a reference to the shared queue.

Ex3: a) Is the main thread waiting for both threads to finish?

b) How do you make the main thread wait for both threads to finish? Hint: You will need to implement the waiting in the main method of Factory, then update the code in both threads to finish instead of running forever?

c) Can the three threads in this implementation communicate? Such as knowing the state of each other, changing the state of each other, synchronise access to the shared channel?