Operating Systems	**CSI3131** ***Assignment 2***	*Winter 2011*

Semaphores/inter-thread synchronization

You must submit your assignment on-line with Virtual Campus. This is the only method by which we accept assignment submissions. Do not send any assignment by email. We will not accept them. We are not able to enter a mark if the assignment is not submitted on Virtual Campus!

The deadline date is firm since you cannot submit an assignment passed the deadline. You are responsible for the proper submission of your assignments and you cannot appeal for having failed to do so. A mark of 0 will be assigned to any missing assignment.

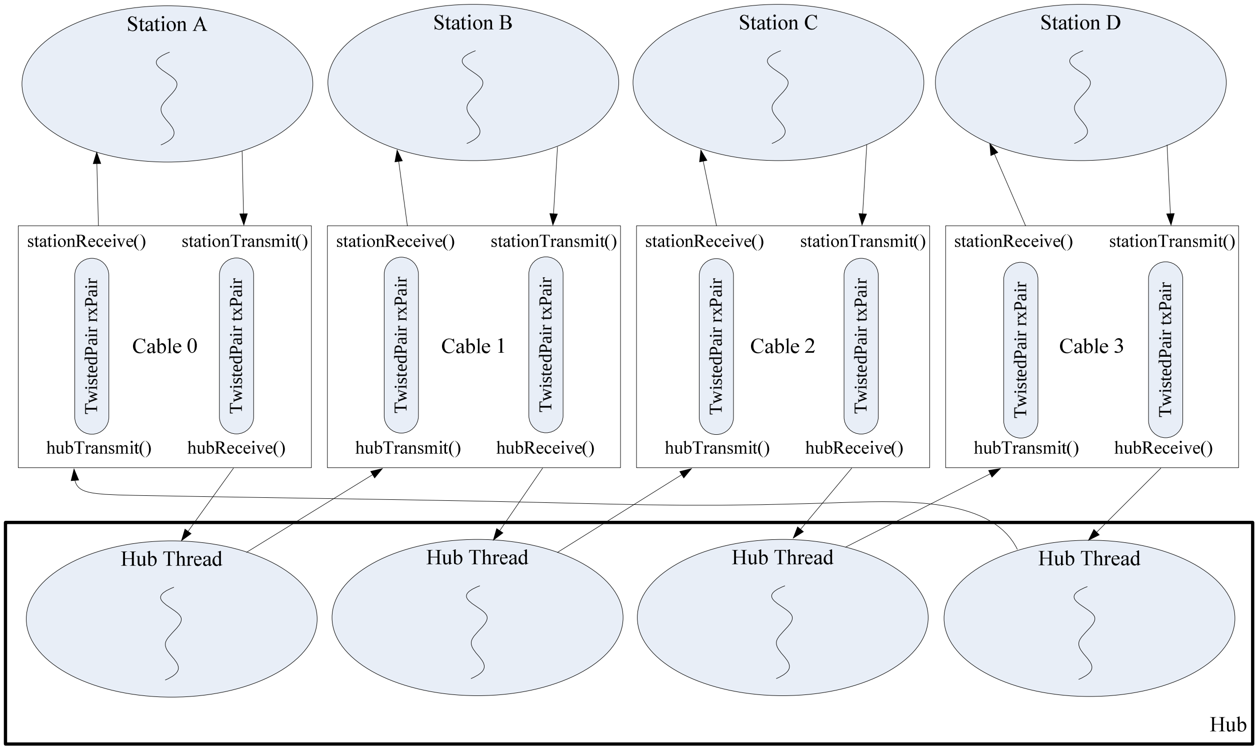
Assignments must be done individually. Any team work, and any work copied from a source external to the student (including solutions of past year assignments) will be considered as an academic fraud and will be forwarded to the Faculty of Engineering for imposition of sanctions. Hence, if you are judged to be guilty of an academic fraud, you should expect, at the very least, to obtain an F for this course. Note that we will use sophisticated software to compare your assignments (with other student's and with other sources...). This implies that you must take all the appropriate measures to make sure that others cannot copy your assignment (hence, do not leave your workstation unattended).

Goal: Practise semaphore usage.

Marks:	50
Posted:	Feb 16, 2009
Due:	March 13, Midnight

Description (Please read the complete assignment document before starting.)

The simulation software from Assignment 1 (simulation of a token ring network) has been translated from a C program to a Java program (see the provided software) that uses threads to simulate each of the Stations. There are also four threads used for monitoring each receive twisted pair in the hub. The organization of threads is illustrated in the following diagram:



The station threads shown in the figure execute the `Station Class` (which extends the `Thread Class`). The hub threads shown in the figure executes the `HubThread Class` (which extends the `Thread Class`). Each of the `Station` and `HubThread` objects contain references to the appropriate `Cable` objects. This means that the `Cable` objects (and referenced `TwistedPair` objects) are common resources to multiple threads. The `Hub Class` (which contains `main()`) creates all threads, allows them to run for 5 seconds, and then terminates the threads. See the provided Java code for more details.

A token ring cable is simulated with the `Cable Class` (four `Cable` objects are used to represent each of the cables from the stations to the hub as shown in the above figure). In the `Cable Class`, two `TwistedPair` objects referenced by `txPair` and `rxPair` are used to simulate the transmit twisted pair and the receive twisted pair respectively (in fact there are two versions of this class, `TwistedPairVer1` and `TwistedPairVer2`). The `TwistedPair` class uses a `String` object (referenced by `String buf`) to represent the data being transmitted across the twisted pair; in other words, the `buf` object represents the buffer used in the pipe of the C project. To make the problem more interesting (and even more realistic), there is a limit placed on the length of the string that a `TwistedPair` object can reference (defined by `maxBufLength`). Four methods have been provided in the Cable class for transmitting to and receiving from the `TwistedPair` objects:

- For communication over the `txPair` object:
  - `stationTransmit()`: this method is called by the station thread to “transmit” across `txPair` that is, adding a frame to the string referenced in `txPair`. This method calls the `xmit` method of the `txPair` object.
  - `hubReceive()`: This method is called by only one of the hub threads, the thread responsible for listening on `txPair`. This method calls the `recv` method of the `txPair` object. The hub thread blocks in `recv` when `txPair` references an empty string `“”`). Otherwise, it returns the String reference in `txPair` to the calling method (and then has `txPair` reference an empty string).
- For communication over the `rxPair` object:
  - `stationReceive()`: This method is called by a station thread to receive frames from `rxPair`. It calls the `recv` method of the `rxPair` object which returns the `String` referenced in `rxPair` by `buf` (and then has `buf` in `rxPair` reference an empty string). The station thread blocks when `buf` references an empty string.
  - `hubTransmit()`: This method is called by hub threads to “transmit” across the `rxPair` that is, adding frames to the string referenced by `rxPair`. This method calls the `xmit` method of the `rxPair` object.

The Challenges

You will note that the provided `TwistedPairVer1` and `TwistedPairVer2` classes are sparse and contains only the critical sections for each of the two methods `xmit()` and `recv()`. You shall complete the two versions with synchronization code.

`TwistedPairVer1`**)** The first version is to be used in the Token Ring application (*TokRing.zip*). In this case only two threads access each of the `TwistedPairVer1` objects. In addition the nature of the Token ring operation means that only one twisted pair is used at anyone time. Thus most threads are blocked trying to read from a `TwistedPairVer1` object. It can also be assumed that when writing to the `TwistedPairVer1` object there is always enough room to write the string (i.e. the limit placed on the length of the `String` is not used in this version). Thus synchronisation is relatively simple. The Java monitor can be used to provide synchronization within the `TwistedPairVer1` class. Use `synchronized`, `wait()`, and `notify()` to develop your solution and complete the source code *TwistedPairVer1.java*.

`TwistedPairVer2`**)** A second project (*TestTp.zip*) is provided to use the `TwistedPairVer2` object with many writers and many readers; notice that there are four writer threads and four reader threads created in this project that all try to access a single `TwistedPairVer2` object. In this case the `TwistedPairVer2` object resembles the *pipe* and needs to provide the following functionality:

- When a reader calls `recv`, the whole `String buf` is returned (i.e. the buffer is emptied).
- A reader thread blocks in recv when the `String buf` references an empty string.
- A writer thread blocks in `xmit` when the string to be added would have `buf` exceed `maxBufLength`; otherwise the string is appended to the `buf`.
- When string is added to the buffer by a writer thread (i.e. when `xmit()` is called), and reader threads are blocked, a reader thread must be unblocked; note that it may be possible for a reader thread to be blocked again.
- When the buffer is emptied by a reader (i.e. when `recv` is called), and writers are blocked, all writers are unblocked; note that it may be possible for writers to become blocked again.

Your challenge, in this version, is to add the required Semaphores and other variables to control the access to the critical sections in `xmit` and `recv`. Do not change other Java files, only the files *TwistedPairVer2.java*.Your solution must NOT contain any *busy wait loops*.

- Use the semaphores to block the threads under the conditions described above.
- Ensure that your solution deals with all situations, that is, block on both receiving and transmitting data.
- Test your solution by running your program many times to get expected (although different) output each time the software is run.
- Trace the execution of your program using `System.out.println` statements to ensure that all parts of your solution have been executed successfully. Note that the attribute "`int tpId`" is provided so that you can include the cable number in any printed messages. Infact the method `logMsg()` is provided to print messages prefixed with the `tpId` and thread identifier.
- In addition to the `Semaphore` class constructor and variables, you may use only the following `Semaphore` methods for synchronization: *acquire* and *release*.
- Do use variables to count the number of reader and writer threads that are blocked.
- **Your solution for the second version should not use any other Java synchronization primitives (for example, you should not use the monitor synchronized, wait(), and notify()).**

To submit your solution:

Submit both files *TwistedPairVer1.java* and *TwistedPairVer2.java* containing your solutions. You will get partial marks for partial solutions. Be sure to include your name and student number at the start of each file.