CSE 522
Homework 2: Threads

Assigned: Sept 22, 2017 (Part 1)
Sept 27, 2017 (Part 2)
Due: October 4, 2017 (11:59 pm)

(Part 2 - clarification and correction noted in red on October 1)

Part 1: Generalized Producer-Consumer

Generalize the Producer-Consumer program of Lecture 7 so that the `data` field in class `DropBox` is an array of `n` integers rather than a single integer. As before, the `Producer` and `Consumer` are concurrent threads that communicate with `DropBox` by executing `put` and `get` operations respectively. Thus, when the dropbox array is empty, we can allow up to `n` consecutive `put` operations without any intervening `get` operation. And, when the dropbox array is full, we can allow up to `n` consecutive `get` operations without any intervening `put`.

The classes `Producer` and `Consumer` as well as method `main` are given to you in the file `ProducerConsumer.java`. Your task is to write the code for class `DropBox`. The key idea is to treat the dropbox array as a *circular buffer.* Proceed as follows.

1. The size, `n`, of the array should be given as a parameter of the constructor of class `DropBox` and kept as a private field of the class.
2. Maintain in class `DropBox` two private fields, `int p` and `int g`, which give, respectively, the index in the array where the next value should be placed by the producer and taken by the consumer. Since the array is treated as circular buffer, these indices should be reset to 0 when they reach the end of the array.
3. Also maintain in class `DropBox` a private field, `int count`, which keeps track of how many values may be taken out by the consumer without any intervening `put` operation by the producer.
4. Define two Boolean functions, `empty()` and `full()`, indicating the status of the dropbox.
5. Write the definitions of the `get()` and `put(int v)` methods of class `DropBox` taking the above features into account.

Run your program and save the object and sequence diagrams generated by JIVE in files named `PC_obj.png` and `PC_seq.png`. Use the "Stacked with Tables" option for the object diagram. Also, the `put()` and `get()` operations should print out on Console the value that they put into and got out of the dropbox respectively.

(a) Explain with reference to the sequence diagram: At which point did the producer have the wait because the dropbox was full? Name the specific call on `put()` when this happened.
(b) Explain with reference to the state diagram: What is the smallest set of fields in class `DropBox` that you would track in order to prove that the dropbox became full? Save your state diagram in file `PC_state.png`.

Write the above explanations in a file called `explain.pdf`.

What to Submit: Prepare a top-level directory named *HW2_Part1_UBITId* where *UBITId* is your UBIT id. In this directory, place your source code `ProducerConsumer.java` and your diagrams `PC_obj.png`, `PC_seq.png` and `PC_state.png` as well as `explain.pdf`. Compress the directory and submit the compressed file using the `submit_cse522` command.

Part 2: Semaphores

Refer to the program `Readers_Writers_with_Priority.java`. This program enforces the basic requirement of the Readers-Writers problem, namely, that multiple `read()` operations on the database may execute concurrently but a `write()` operation must execute in mutual exclusion of any other `write()` or `read()` operation. In addition, it gives priority for `Writer` threads, i.e., a waiting `Writer` thread gets to execute the `write()` operation ahead of any waiting `Reader` thread.

(a) The program has a bug which you should correct. Run the program through JIVE and observe the object diagram in order to understand the cause of the bug. Name your corrected file as `Readers_Writers_Corrected.java`. Run it through JIVE and make sure that the program is behaving correctly. Provide an explanation of the bug in a file `explain.pdf`.

(b) Translate your *corrected program* using semaphores using the method discussed in Lecture 9 for translating synchronized methods and the wait-notify constructs. Name the file as `Readers_Writers_Semaphores.java`. In developing your translation, you should maintain separate queues for waiting readers and waiting writers and also ensure that waiting threads are not unnecessarily notified.

Generate different state diagrams to show that your program is working correctly:

(i) A state diagram for `Database:1->r` and `Database:1->w`. Name this diagram as `RW1.png`. This helps to check the basic readers-writers requirement: `r == 0` when `w == 1`, and also that `r >= 0` when `w == 0`.

(ii) A state diagram for `Database:1->r`, `Database:1->w`, and `Database:1->data`. Name this diagram as `RW2.png`. It shows in more detail how the database was updated.

(iii) A state diagram for `Database:1->wr` and `Database:1->ww`. Name this diagram as `RW3.png`. It helps to check that priority is correctly implemented: the value of `wr` should never decrease when `ww > 0`. (The field `wr` maintains a count of the number waiting readers and `ww` maintains a count of waiting writers.) Note that this property does not hold for `Readers_Writers_with_Priority.java`.

What to Submit: Prepare a top-level directory named *HW2_Part2_UBITId* where *UBITId* is your UBIT id. In this directory, place your source codes `Readers_Writers_Corrected.java` and `Readers_Writers_Semaphore.java`; and also place your diagrams `RW1.png`, `RW2.png` and `RW3.png;` as well as your explanation file, `explain.pdf`. Compress the directory and submit the compressed file using the `submit_cse522` command.

End of Homework #2