LAB FOR AUTOMATED REASONING AND ANALYSIS LARA

CONCURRENT BOUNDED BUFFER

Assignment handout, updated 11:00am April 7, 2016: See the Grading Infrastructure page for setting up your repository.

The goal of this assignment is to implement a fixed size buffer that can be accessed concurrently by multiple threads running in parallel. This is a simple yet important data structure that is widely used in concurrent applications. In particular, we provide you an implementation of a Publish/Subscribe (PubSub) application that uses the bounded buffer you would be developing. We will provide a brief overview of the Publish/Subscribe application, and instructions on how to run it towards the end of this assignment. However, the only data structure that you should implement for this assignment is the bounded buffer, which is explained below:

Bounded Buffer - Definition and Operations

A concurrent bounded buffer is a data structure that supports the following two operations:

- a method `put(e)` that puts a value `e` in the buffer,
- a method `take` that takes a value from the buffer, and removes it.

Importantly, the `put` and `take` methods should follow the FIFO (First-In, First-Out) semantics. Intuitively, this means that `take` should return the *oldest* element that was `put` into the buffer. (A remark: the FIFO semantics in the presence of concurrency is formally defined by notions like Linearizability. But, for this assignment, it suffices to intuitively understand the expected behavior of the buffer based on the description presented here.)

The buffer is *bounded*, which means that it can only store a finite number of elements. The maximum number of elements the buffer can hold is passed as an argument to the constructor of `BoundedBuffer`. The following is the definition of the data structure, which can be found in the file: `src/main/scala/pubsub/collection/BoundedBuffer`

```
class BoundedBuffer[T](size: Int) extends AbstractBoundedBuffer[T](size) {
  override def put(e: T): Unit = ???
  override def take(): T = ???
}
```

A `put` operation invoked on a buffer that is full will *wait* (or block) until an element is taken from the buffer. Dually, a `take` operation invoked on an empty buffer will *wait* until an element is put into the buffer. Note that mutiple `put` and `take` operations can be initiated concurrently from different threads. Hence, care must taken to ensure that the `buffer` is not accessed or left in an inconsisitent state by the `put` or `take` methods. To understand the challenges better let us start by considering a sequential setting.

A Sequential Bounded Buffer

Let us try to implement the `BoundedBuffer` class without worrying at all about concurrency. That is, imagine that there is only one thread that performs a sequence of `put` and `take` operations on a object of type `BoundedBuffer`. Can you implement the `put` and `take` methods in this setting ? Note that your implementation should run in constant time in the worst-case. (A remark: the data structure you implement in this case is also called a *circular queue*.)

To write your solution, use the `pubsub.collection.BoundedBuffer` class skeleton that is given to you in the handout. You are given two *variables*: `count` and `head`, and an internal buffer that exposes a minimal array-like API. It is important that you use the variables given to you, and the internal buffer API. *You are strongly advised not to create any other fields or classes or objects*. Otherwise, you may fail the testcases.

The following is an illustration of the input/output behavior of the buffer in the sequential case. In the comments, we show the contents of the array used internally by the `BoundedBuffer` class, and the return values (if any) of the operations.

```
def example1() = {                  // Internal Buffer:
  val buf = new BoundedBuffer(2)    // |   |   |
  buf.put(1)                        // | 1 |   |
  buf.put(2)                        // | 1 | 2 |
  val x = buf.take()                // |   | 2 | and x is 1
  buf.put(3)                        // | 3 | 2 |
  val y = buf.take()                // | 3 |   | and y is 2
}
```

Now, think about what should happen if we try to `put` an element into a buffer that is full ? Obviously, the element cannot be stored in the buffer. Furthermore, there is no use of waiting for some space to be created in the buffer because no element in the buffer could be read simulatenously. (Recall that there is only one thread accessing the buffer in the sequential setting). Hence, you may throw an exception in this case. Similarly, handle the dual case of invoking a `take` on an empty buffer.

Think about what happens when your sequential buffer is used in a concurrent system. What can happen? Try running the test suite (`sbt test`) that was given to you with the handout. What can you see?

As you may have seen, the test suite was able to find many problematic schedules for the non-concurrent buffer. This is to be expected! Now, using your sequential implementation as a stepping stone, let us develop a concurrent bounded buffer, which is your real goal.

Concurrent Bounded Buffer

Now, let us imagine that there are two threads currently performing operations on a shared buffer of max size 2 as shown below:

```
val buf = new BoundedBuffer(2)
  Thread 1          Thread 2
  buf.put(1)        y = buf.take()
```

The first change you need to make to your sequential implementation is that `buf.take()` invoked by `Thread 2` should *wait* until an element is put into the buffer by `Thread 1`. Similarly, a `put` method should *wait* until an element is taken from the buffer. Throwing an exception in these cases as in the sequential case is not acceptable here because operations can be happening simultaneously. In other words, a `put` that cannot succeed now can succeed in the future, because of a concurrently executing `take`.

For this purpose you may use a `wait` method of the Java standard library. You can find a brief description of the semantics of the method, and references for further reading in the section *Synchronization Primitives* below. An alternative is to loop continously until the buffer has an element. This is called as *busy waiting*. But as you can imagine this wastes precious processor cycles, and hence is not preferred. (See the section on busy waiting as well for some examples.)

With this waiting strategy, in the above example, `Thread 2` will enter into *wait* state if it executes before `Thread 1`, and `Thread 1` will proceed to execute. However, after completing the execution `Thead 1` should *wake up* `Thread 2` (and also every other thread that would be waiting on the buffer to become non-empty). For this purpose, you have to use the `notify()`, and/or `notifyAll()` primitives (see section *Synchronization Primitives*). Note that there are subtle differences between `notify()` and `notifyAll()` which are also breifly explained below.

Important Hint: You should invoke the synchronization primitives like `wait`, `notify`, `synchronized` etc. only on the `this` reference when implementing the methods of the `BoundedBuffer` class. Otherwise, your implementation may get stuck (and run into deadlocks) while testing

Also remember that `notify` and `wait` should be invoked only inside a `synchronized` block.

Now that we know how to implement the waiting strategy to alleviate the above problem of `Thread 2` running before `Thread 1`, let us look at other issues that may arise when `put` and `take` methods run concurrently. In particular, say `Thread 1` starts before `Thread 2`. Let us study how the operations they perform can interleave, assuming that you implemented them using `count` and `head`. Consider the following interleaving of operations of `put` and `get`

```
val buf = new BoundedBuffer(2)
  Thread 1                    Thread 2
  //start buf.put(1)
                              //start buf.take
  //reads count -> 0
  //...
  //updates count -> 1
                              //reads count -> 1
                              //read internal buffer(0)  (crash !!)
  //internal buffer(0) <- 1
```

Clearly, if your code permits the above interleaving then it would crash. Now, your goal is to come up with adequate synchronization so that no interleaving of operations of `put` and `take` performed by any number of threads result either in a crash or in an incorrect buffer state. You may use all the synchornization primitives described below.

The test suite provided along with the project may help you find problematic schedules. A schedule is a sequence of operations carried out by the different threads such as the one shown above. The test suite runs your code on several thousand schedules. You may use that to check whether your bounded buffer reaches an consistent state on some schedule.

An incorrect run on a schedule could either be that: - An operation returns an incorrect value. - An operation throws an exception when it should not.

Remember that some operations, even through they look *atomic*, are decomposed into multiple smaller operations. For instance, consider `buffer(head) = ...`. Is this atomic ? In the remaining sections, we talk about the synchornization primitives and some of the commons cases of errors.

Synchronization Primitives

- `synchronized[T](body: => T)`, which executes `body` in a critical section. This ensures that only a single thread at a time can execute the instructions passed as argument.
- "wait()", which stops the execution of the thread until it is notified. `wait()` can only be called within a `synchronized` block.

- ''notify()'', which unblocks one of the waiting thread. Note that it's not possible to specify which thread is unblocked.
- ''notifyAll()'', which unblocks all of the waiting threads.

Common Pitfalls

Busy Waiting

Consider the following code snippet that uses a while loop (with an empty body) to guard the execution of the `put` and `take` operations.

```
class BoundedBuffer[T](size: Int) extends AbstractBoundedBuffer[T](size) {
  override def put(e: T): Unit = {
    while (isFull) {}
    // Rest of your code
    ???
  }
  override def take(): T = {
    while (isEmpty) {}
    // Rest of your code
    ???
  }
}
```

What is the problem with this way of waiting? Apart with those potential problems, is this code behaving correctly in a concurrent setting i.e, is there a schedule that would result in an incorrect behavior on the above code. *Hint: what happens when one `put` and two `take` are running in parallel ?*

notify() vs notifyAll()

As per the documentation documentation:_ of `notify()`, invoking the `notify` method on a object `o` wakes up a *single* thread that is waiting on the object `o`'s monitor. If there are multiple threads waiting on `o`, one of them is chosen **arbitrarily** at the discretion of the implementation. Note that a thread waits on an object's monitor by calling one of the wait methods.

Using the `notify` method often leads to tricky problems which can sometimes result in deadlocks. This can lead to bugs that can be hard to see and reproduce. For this reason, it is often strongly advised to use `notifyAll` instead. Can you figure out why this is the case? *Hint: Could it ever happen that the thread unblocked by notify somehow cannot execute even though there are other threads that could execute if they were unblocked instead ?*

Waiting using if vs while

Due to the semantics of `notify` and `notifyAll`, there are no guarantees that the condition on which a thread is waiting will be fullfilled when the thread is unblock. Therefore, the following code is problematic:

```
synchronized {
  if (!condition) {
    wait()
  }
  // We have no garantees that the condition holds here.
}
```

Instead, the waiting thread should always recheck the condition when it is unblocked. This can easily be done using a `while` instead of `if`.

Can you come up with a schedule in which using a simple `if` would be problematic?

Hint: Try using a system with 3 threads, two of which try to execute `take` and one of which executes `put` in an initially empty buffer?

Hint: Note that, as specified in the documentation of ''notify'': > The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, **the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object**.

Using the Pub/Sub Server

The following section describes how you can run and use the Pub/Sub server implemented for you, This small server allows any amount of people to connect and chat together around topics of their choice. Playing with it may help you discover errors in your ConcurrentBoundedBuffer.

Running the server

You can start the Pub/Sub Server as any normal SBT project, simply type `sbt run` while in the root folder of the project. No additionnal setup is required.

Connecting to the server

You can connect to the server via command line interface and the telnet protocol. When the server is started it will display a line such as:

```
Connect to <server_ip> (or `localhost`), port 7676 with `telnet` to join this server
```

Which gives you the IP address of the running server. In your command line you must do the following telnet command: `telnet <server_ip> 7676` You should then be greeted by the server with a `connection_ack`.

telnet is not enabled by default on windows 8 and 10. here are fast and simple instructions to enable telnet on recent windows.

Commands availiable

Now that the server is running and you have a connected client, here is the comprehensive list of the commands any client can use on the server:

- `rename <nickname>`: change your display name to `<nickname>`. Your new name cannot contain any space or single quote.
- `subscribe <topic>`: add yourself to the `<topic>` subscriber list. You will recieve all messages sent to that topic.
- `unsubscribe <topic>`: remove yourself from the `<topic>` subscriber list. You will no longer recieve messages sent to that topic.
- `publish <topic> '<message>'`: publish the `<message>` of your choice to the `<topic>` of your choice. Your `<message>`can span on mutliple lines but has to be enclosed with single quote `'` to be deemed valid.
- `leave`: end the connection with the server in a polite manner

parcon16/problem_statement_and_handout.txt · Last modified: 2016/04/11 18:48 by ravi