Read Write Lock

We discuss a common interview question involving synchronization of multiple reader threads and a single writer thread.

Read Write Lock

> Imagine you have an application where you have multiple readers and a single writer. You are asked to design a lock which lets multiple readers read at the same time, but only one writer write at a time.

Solution

> First of all let us define the APIs our class will expose. We'll need two for writer and two for reader. These are:
>
> - acquire_read_lock
> - release_read_lock
> - acquire_write_lock
> - release_write_lock
>
> This problem becomes simple if you think about each case:

1. Before we allow a reader to enter the critical section, we need to make sure that there's no **writer** in progress. It is ok to have other readers in the critical section since they aren't making any modifications.

2. Before we allow a writer to enter the critical section, we need to make sure that there's **no reader or writer** in the critical section.

Let's start with the reader use case. We can have multiple readers acquire the read lock and to keep track of all of them; we'll need a count. We increment this count whenever a reader acquires a read lock and decrement it whenever a reader releases it.

Releasing the read lock is easy, but before we acquire the read lock, we need to be sure that no other writer is currently writing. Again, we'll need some variable to keep track of whether a writer is writing. Since only a single writer can write at a given point in time, we can just keep a boolean variable to denote if the write lock is acquired or not.

Additionally, we'll also need a condition variable for the readers and writers to wait while the other party is in progress. We'll use the associated lock with the condition variable to guard the sections of the code where we manipulate any shared variables. Let's translate what we have discussed so far into code.

```
class ReadersWriteLock():

    def __init__(self):
        self.cond_var = Condition()
        self.write_in_progress = False
        self.readers = 0

    def acquire_read_lock(self):
        pass

    def release_read_lock(self):
        pass

    def acquire_write_lock(self):
        pass

    def release_write_lock(self):
        pass
```

Next we'll take a stab at the **acquire_read_lock()** method.

```
    def acquire_read_lock(self):

        self.cond_var.acquire()
        while self.write_in_progress is True:
            self.cond_var.wait()

        self.readers += 1
        self.cond_var.release()
```

Note that multiple reader threads can invoke the above method and not block if no write is in progress thus satisfying the condition that multiple readers can read at the same time. Next let's examine the **release_read_lock()** method.

```
    def release_read_lock(self):
        self.cond_var.acquire()
        self.readers -= 1
        self.cond_var.notifyAll()
        self.cond_var.release()
```

When releasing the read lock, each reader decrements the **readers** count. Since multiple reader threads can modify the shared readers' count, we guard it with the condition variable's implicit lock. Also, we notify any writer threads waiting on the condition variable. Note that a reader notifies the condition variable even if the readers' count is not zero. We can improve the code as follows, but the above implementation will also work.

```
        self.cond_var.acquire()
        self.readers -= 1
        if self.readers == 0:
            self.cond_var.notifyAll()
        self.cond_var.release()
```

Now let's turn to the writer thread case. A writer should only proceed if no reader or writer is currently in progress, implying we need to check for two conditions. The implementation is as follows:

```
    def acquire_write_lock(self):
        self.cond_var.acquire()

        while self.readers is not 0 or self.write_in_progress
is True:
            self.cond_var.wait()
        self.write_in_progress = True

        self.cond_var.release()
```

The last piece is the **`release_write_lock()`** which simply sets the **`write_in_progress`** to false and notifies any reader threads waiting on the condition variable.

```
    def release_write_lock(self):
        self.cond_var.acquire()
        self.write_in_progress = False
        self.cond_var.notifyAll()
        self.cond_var.release()
```

The complete code appears below in the code widget.

```
from threading import Condition
from threading import Thread
from threading import current_thread
import time
import random


class ReadersWriteLock():

    def __init__(self):
        self.cond_var = Condition()
        self.write_in_progress = False
        self.readers = 0

    def acquire_read_lock(self):
        self.cond_var.acquire()

        while self.write_in_progress is True:
            self.cond_var.wait()

        self.readers += 1

        self.cond_var.release()

    def release_read_lock(self):
        self.cond_var.acquire()

        self.readers -= 1
        if self.readers is 0:
            self.cond_var.notifyAll()

        self.cond_var.release()

    def acquire_write_lock(self):
        self.cond_var.acquire()

        while self.readers is not 0 or self.write_in_progress is True:
            self.cond_var.wait()
        self.write_in_progress = True

        self.cond_var.release()

    def release_write_lock(self):
        self.cond_var.acquire()

        self.write_in_progress = False
        self.cond_var.notifyAll()

        self.cond_var.release()
```

```
def writer_thread(lock):
    while 1:

        lock.acquire_write_lock()
        print("\n{0} writing at {1} and current readers = {2}".format(current_threa
                                                                        lock.readers)

        write_for = random.randint(1, 5)
        time.sleep(write_for)
        print("\n{0} releasing at {1} and current readers = {2}".format(current_thr
                                                                          lock.reader

              flush=True)
        lock.release_write_lock()
        time.sleep(1)


def reader_thread(lock):
    while 1:
        lock.acquire_read_lock()
        print("\n{0} reading at {1} and write in progress = {2}".format(current_thr
                                                                          lock.write_

        read_for = random.randint(1, 2)
        time.sleep(read_for)
        print("\n{0} releasing at {1} and write in progress = {2}".format(current_t
                                                                            lock.writ

        lock.release_read_lock()
        time.sleep(1)


if __name__ == "__main__":

    lock = ReadersWriteLock()

    writer1 = Thread(target=writer_thread, args=(lock,), name="writer-1", daemon=Tr
    writer2 = Thread(target=writer_thread, args=(lock,), name="writer-2", daemon=Tr

    writer1.start()

    readers = list()
    for i in range(0, 3):
        readers.append(Thread(target=reader_thread, args=(lock,), name="reader-{0}"

    for reader in readers:
        reader.start()

    writer2.start()

    time.sleep(15)
```

The above simulation runs for 15 seconds and from the output, you can verify that a writer thread gets to exclusively write. The acquire and release statements for a writer thread must appear consecutively to prove the correct working.

Follow-up Question

Since only a single writer thread can be active at any given point in time, we can also design the writer APIs as follows:

```
    def acquire_write_lock(self):
        self.cond_var.acquire()

        while self.readers is not 0 or self.write_in_progress
is True:
            self.cond_var.wait()
        self.write_in_progress = True

    def release_write_lock(self):
        self.write_in_progress = False
        self.cond_var.notifyAll()
        self.cond_var.release()
```

The acquisition and release of the condition variable is split across two methods. The code works just as fine as shown in the code widget below. However, can you think of why this might be a bad idea? Read the explanation which appears after the code widget.

```
from threading import Condition
from threading import Thread
from threading import current_thread
import time
import random


class ReadersWriteLock():

    def __init__(self):
        self.cond_var = Condition()
        self.write_in_progress = False
        self.readers = 0

    def acquire_read_lock(self):

        self.cond_var.acquire()
        while self.write_in_progress is True:
            self.cond_var.wait()

        self.readers += 1
        self.cond_var.release()

    def release_read_lock(self):
        self.cond_var.acquire()
        self.readers -= 1
        if self.readers == 0:
            self.cond_var.notifyAll()
        self.cond_var.release()

    def acquire_write_lock(self):
        self.cond_var.acquire()

        while self.readers is not 0 or self.write_in_progress is True:
            self.cond_var.wait()
        self.write_in_progress = True

    def release_write_lock(self):
        self.write_in_progress = False
        self.cond_var.notifyAll()
        self.cond_var.release()


def writer_thread(lock):
    while 1:
        lock.acquire_write_lock()
        print("\n{0} writing at {1} and current readers = {2}".format(current_threa
                                                                      lock.readers)
        write_for = random.randint(1, 5)
        time.sleep(write_for)
        print("\n{0} releasing at {1} and current readers = {2}".format(current_thr
                                                                            lock.reader
              flush=True)

        lock.release_write_lock()
        time.sleep(1)


def reader_thread(lock):
    while 1:
        lock.acquire_read_lock()
        print("\n{0} reading at {1} and write in progress = {2}".format(current_thr
                                                                            lock.write_
        read_for = random.randint(1, 2)
        time.sleep(read_for)
        print("\n{0} releasing at {1} and write in progress = {2}".format(current_t
                                                                              lock.writ
        lock.release_read_lock()
        time.sleep(1)


if __name__ == "__main__":

    lock = ReadersWriteLock()

    writer1 = Thread(target=writer_thread, args=(lock,), name="writer-1", daemon=Tr
    writer2 = Thread(target=writer_thread, args=(lock,), name="writer-2", daemon=Tr

    writer1.start()

    readers = list()
    for i in range(0, 3):
        readers.append(Thread(target=reader_thread, args=(lock,), name="reader-{0}"

    for reader in readers:
        reader.start()

    writer2.start()

    time.sleep(20)
```

The above approach might seem more efficient since a writer thread only acquires and releases the condition variable once during operation. Also, we can eliminate the **`self.write_in_progress is True`** condition in the while loop of the **`acquire_write_lock()`** method. However, the Achilles' heel of the above solution is that if the writer thread dies

> between the two method calls, the entire system would enter a deadlock!

← Back

Implementing Semaphore

Next →

Unisex Bathroom Problem

☑ Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/read-write-lock__interview-practise-problems__python-concurrency-for-senior-engineering-interviews)