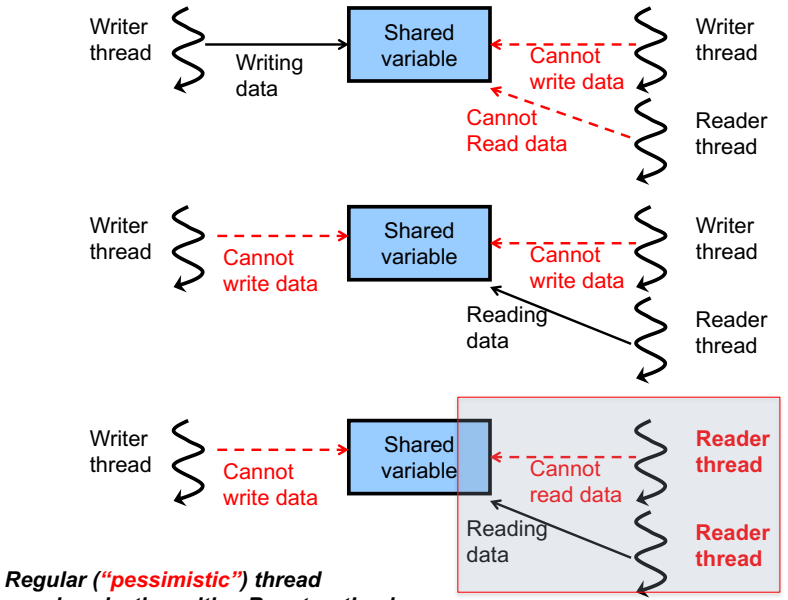
Optimistic Locking with Read-Write Locks

Read-Write Locks

- Regular lock (`ReentrantLock`)
  - To avoid race conditions by guarding a variable shared by multiple threads.
- Read-Write lock
  - A slight extension to `ReentrantLock`
  - A bit more *optimistic* than a regular lock to seek performance improvement.
    - `java.util.concurrent.locks.ReentrantReadWriteLock`

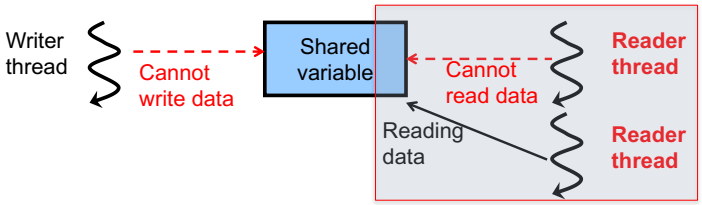
Room for Performance Improvement?

- Locking is often computationally expensive.
  - It takes some time to acquire/release a lock.
  - A thread does nothing while it is in the Blocked state.
- Where to gain performance improvement?
  - When you have multiple "reader" threads that read data from a shared variable, do we have to synchronize (i.e., mutually exclude) them?
    - No, as far as the value of the shared variable never changes.
    - We can be *optimistic* NOT to synchronize "reader" threads.

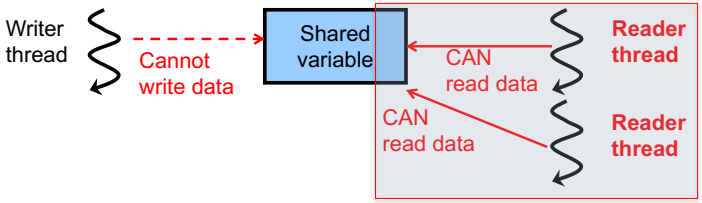


Regular ("pessimistic") thread synchronization with a ReentrantLock

Do we really need to synchronize (i.e. mutually exclude) these readers?



"Pessimistic" thread synchronization with a ReentrantLock



"Optimistic" thread synchronization

ReentrantReadWriteLock

```
public class ReentrantReadWriteLock implements ReadWriteLock{

    public class ReentrantReadWriteLock.ReadLock
        implements Lock{}

    public class ReentrantReadWriteLock.WriteLock
        implements Lock{}

    public ReentrantReadWriteLock.ReadLock readLock(){}
    public ReentrantReadWriteLock.WriteLock writeLock(){}
}
```

- Provides two locks
  - As inner *singleton* classes
    - Both implement the `Lock` interface.
  - `ReadLock` for reader threads to read data from a shared variable.
  - `WriteLock` for writer threads to write data to a shared variable.
- Provides factory methods for the two locks: `readLock()` and `writeLock()`.

- A reader can acquire a read lock even if it is already held by another reader,
  - **AS FAR AS** no writers hold a write lock.
- Writers can acquire a write lock **ONLY IF** no other writers and readers hold read/write locks.

Can a thread acquire…? \ When another thread holds …?	ReadLock	WriteLock
ReadLock	**Yes**	No
WriteLock	No	No

This turns to be "No" if you use a regular lock.

An Example Optimistic Locking

```
int i; // shared variable
ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
```

- For reading data from a shared variable:

```
rwLock.readLock().lock();
System.out.println(i);
rwLock.readLock().unlock();
```

- For writing data to a shared variable

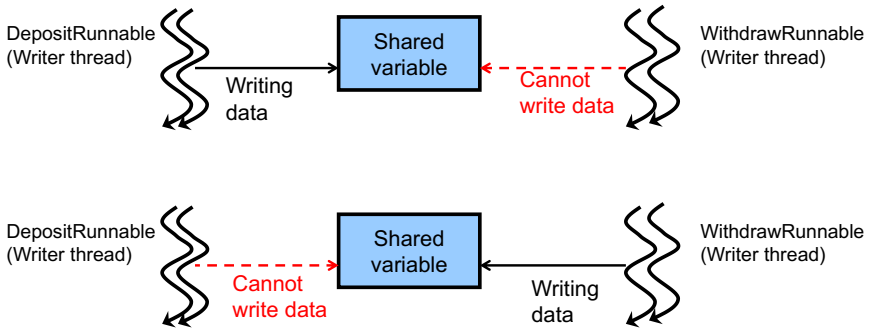
```
rwLock.writeLock().lock();
i++;
rwLock.writeLock().unlock();
```

`ReadLock` and `WriteLock`

- Work similarly to `ReentrantLock`.
  - Support nested locking and thread reentrancy.
  - Support interruption via `Thread.interrupt()`.
- `WriteLock`
  - Returns a `Condition` object when `newCondition()` is called.
- `ReadLock`
  - Throws an `UnsupportedOperationException` when `newCondition()` is called.
    - Reader threads never need condition objects.
    - Readers threads never call `signal()` and `signalAll()`.

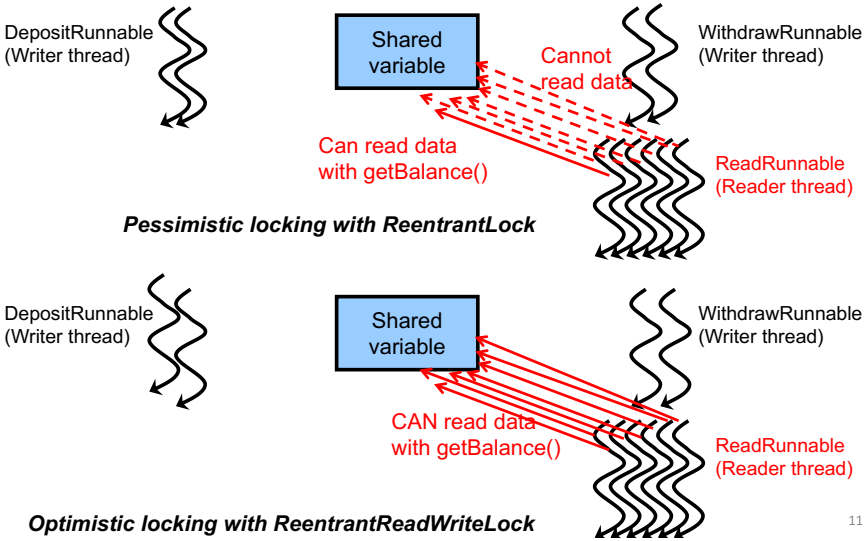
Sample Code

- ThradSafeBankAccount2



Always need pessimistic locking (w/ ReentrantLock) for writer threads

- ThradSafeBankAccount3 and ThradSafeBankAccount4



Pessimistic locking with ReentrantLock

Optimistic locking with ReentrantReadWriteLock

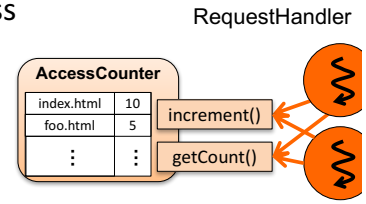
- ThradSafeBankAccount3
  - 43 msec
- ThradSafeBankAccount4
  - 33 msec
    - 23% (10/43) faster
      - thanks to optimistic locking

When to Use Optimistic Locking?

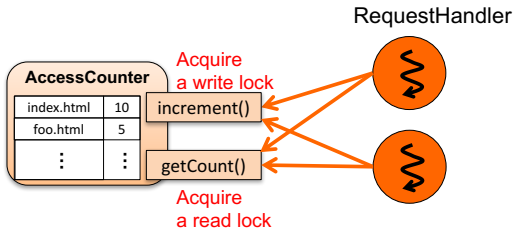
- When many reader threads run.
- When reader threads run more often than writer threads.
- When a read operation requires a long time to be completed.

HW 15

- Recall a previous HW to implement a concurrent access counter, assuming the development of a web server
- **`AccessCounter`**
  - Maintains a `HashMap` that associates a relative file path with its access count.
  - `increment()` accepts a file path and increments the file's access count.
  - `getCount()` accepts a file path and returns the file's access count.
- **`RequestHandler`**: a Runnable class
  - `run()`: Chooses a file at random, calls `increment()` and `getCount()` for that file, and sleep for a few seconds.



- Use a **`ReentrantReadWriteLock`** rather than **`ReentrantLock`**.
  - Have each thread acquire
    - a write lock in `increment()`
    - a read lock in `getCount()`
- Due: May 3 (Thu) midnight



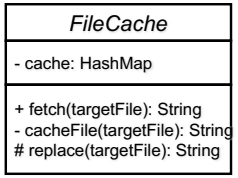
Recap: File Caching in a Web Server

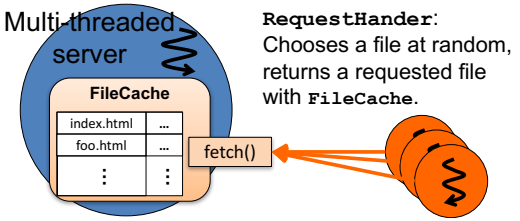
- A web server
  - Receives a connection establishment request from a client (browser).
  - Receives an HTTP command
  - Retrieves a target HTML file from the local disk and returns it to the client.

- File caching
  - Keep frequently-accessed files in the memory space rather than an external/peripheral storage (e.g. HDD).
    - Obtain a requested file from an external storage for the first request
    - Keep (or *cache*) the file in the memory space
    - Use the cached file for the subsequent requests
      - Skip the overhead to access an external storage
  - Can improve the performance (response time and throughput) of a web server.
    - Memory access is much faster than HDD access.

File Caching in a Concurrent Web Server

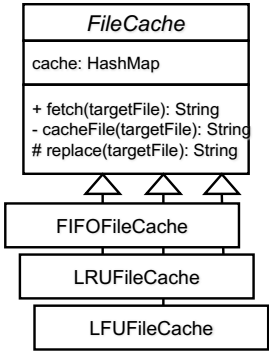
- Assume *thread-per-request* concurrency
- **`FileCache`** (abstract class)
  - Maintains a fixed-sized map that associates a file's relative path with its (string) content.
    - Use **`java.util.HashMap`** and **`java.nio.Path`**
  - **`fetch()`**
    - accepts a file path and returns the content of the file from the **`HashMap`**.
  - **`cacheFile()`**
    - accepts a file path and adds a new path-content pair to the **`HashMap`**.



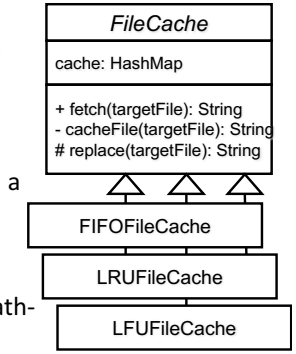


- **`FileCache`** (abstract class)
  - Maintains a *fixed-sized* map, which sets the max number of path-content pairs.
  - Performs *cache replacement* in **`replace()`**.
    - Decides which path-content pairs to keep cached and which ones to be removed
      - when the number of path-content pairs reaches the cap.
  - **`fetch()`**:

```
if( targetFile has been cached )
    return targetFile's (cached) content.
if( cache is not full )
    return cacheFile(targetFile);
else if
    return replace(targetFile);
```



- Different subclasses of **`FileCache`** implement different *cache replacement policies*.
  - First In, First Out (FIFO)
    - Replaces the oldest path-content pair with a new one.
  - Least Frequently Used (LFU)
    - Replaces the least frequently requested path-content pair with a new one.
    - Can take advantage of **`AccessCounter`**
  - Least Recently Used (LRU)
    - Replaces the least recently requested path-content pair with a new one.

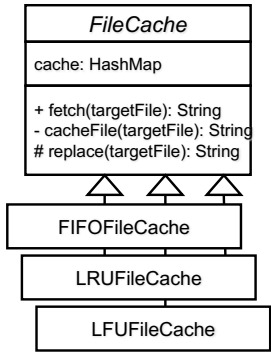


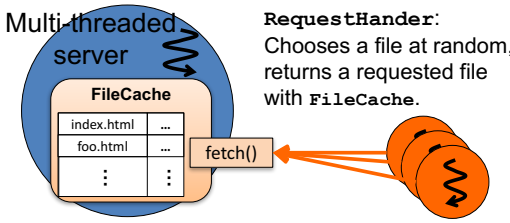
HW 16

- Implement FileCache and its 2 subclasses (FIFO and LFU) in a thread-safe manner.
  - Use a reentrant lock in fetch()
    - Use nested locking in cacheFile() and replace()
- Place some text files
  - FileCache (abstract class)
    FIFOFileCache (extending FileCache)
    LFUFileCache (extending FileCache)
    AccessCounter
    RequestHandler (implementing Runnable)
    <file_root>
    a.html
    b.html
    …
- RequestHandler's run()
  - Chooses a file at random and calls fetch() for that file.
  - Calls AccessCounter's increment() as well.
- main()
  - Creates and starts multiple threads (e.g., 10+ threads) to call FileCache's fetch() and AccessCounter's increment() concurrently.

- Due: May 8 (Tue) midnight

- **`fetch()`**
  - `acquire a lock;`
    `if( targetFile has been cached )`
    `  return targetFile's (cached) content.`
    `if( cache is not full )`
    `  return cacheFile(targetFile);`
    `else if`
    `  return replace( targetFile );`
    `release a lock;`





`RequestHander`: Chooses a file at random, returns a requested file with **`FileCache`**.

Replacing a Regular Lock with a RW Lock

- **`fetch()`**
  - `if( targetFile has been cached )`
    `  return targetFile's (cached) content.`
    `if( cache is not full )`
    `  return cacheFile(targetFile);`
    `else if`
    `  return replace( targetFile );`

Read data from the cache

Write data to the cache

- Use a ***read lock*** in the reading part and a ***write lock*** in the writing part.

- With a regular lock
  -
```
acquire a lock;
if( targetFile is cached )
    return targetFile's content;
if( cache is not full )
    return cacheFile(targetFile);
else if
    return replace(targetFile);
release a lock;
```
- With a read-write lock
  -
```
acquire a read lock;
if( targetFile is cached )
    return targetFile's content;
release a read lock;
acquire a write lock;
if( cache is not full )
    return cacheFile(targetFile);
else if
    return replace(targetFile);
release a write lock;
```

- With a read-write lock
  -
```
acquire a read lock;
if( targetFile is cached )
    return targetFile's content;
release a read lock;
acquire a write lock;
if( cache is not full )
    return cacheFile(targetFile);
else if
    return replace(targetFile);
release a write lock;
```
- Is this thread-safe?
- No. What if a context switch occurs b/w releasing a read lock and acquiring a write lock?

- With a read-write lock
  -
```
acquire a read lock;
if( targetFile is cached )
    return targetFile's content;
release a read lock;
acquire a write lock;
if( cache is not full )
    return cacheFile(targetFile);
else if
    return replace(targetFile);
release a write lock;
```
- Is this thread-safe?

- NOT thread-safe
  -
```
acquire a read lock;
if( targetFile is cached )
    return targetFile's content;
release a read lock;
acquire a write lock;
if( cache is not full )
    return cacheFile(targetFile);
else if
    return replace(targetFile);
release a write lock;
```
A context switch can occur here.
- Thread-safe
  -
```
acquire a write lock;
if ( targetFile is NOT cached ){
  if( cache is not full )
    return cacheFile(targetFile);
  else if
    return replace(targetFile); }
acquire a read lock;
release a write lock;
  return targetFile's content;
release a readlock;
```
No context switch can occur here.
  - ***Lock downgrading***: Acquires a write lock first and then a read lock before releasing the write lock.
    - Lock upgrading is not possible.

With Try-Catch-Finally Blocks…

```
ReentrantReadWriteLock rwlock = new ReentrantReadWriteLock();
rwlock.writeLock().lock();
try{
  if( targetFile is NOT cached ){
    if( cache is not full )
      return cacheFile(targetFile);
    else if
      return replace(targetFile);
  }
  rwlock.readLock().lock(); // downgrading
}
finally{
  rwlock.writeLock().unlock();
}
try{
  return targetFile's content
}
finally{
  rwlock.readLock().unlock();
}
```

HW 17

- Revise your HW 16 code by replacing **`ReentrantLock`** with **`ReentrantReadWriteLock`**.
  - FileCache (abstract class)
  - FIFOFileCache (w/ ReentrantLock)
  - LFUFileCache (w/ ReentrantLock)
  - FIFOFileCacheRW (w/ ReentrantReadWriteLock)
  - LFUFileCacheRW (w/ ReentrantReadWriteLock)
- Due: May 10 (Thu) midnight