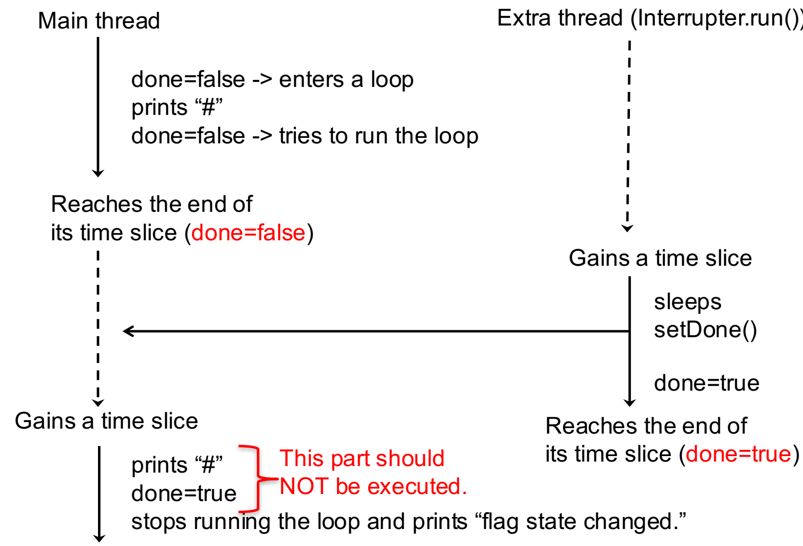
Exercise: ThreadUnsafeFlag.java

- Not thread-safe. Race conditions can occur.
  – Thread safety:
    - No race conditions
    - No deadlocks

```
• public class ThreadUnsafeFlag{
    private boolean done = false;

    public void setDone(){
      done = true;        // writing part
    }

    public void work(){
      while(!done){       // reading part
        System.out.println("#");
      }
    }}
```

A Potential Race Condition



Solution: Use a Lock and Balking

```
– ReentrantLock lock = new ReentrantLock();
  done = false;
  ……
  while(true){
    lock.lock();
    try{
      if(done) break; // balking
    }finally{
      lock.unlock();
    }
    System.out.println(...);
  }

– void setDone(){
    lock.lock();
    try{
      done = true;
    }finally{
      lock.unlock();
    }
  }
```

"Visibility" Issue

- The current (most up-to-date) value of the shared variable "counter" is not *visible* (synchronized) for all threads.

- Locking preserves ***atomicity AND visibility***.

Be VERY Careful

- When multiple threads share and access a variable concurrently.
  - Make sure that a shared variable is protected with a lock.
    - Surround reading and writing parts with lock() and unlock().
      - Reading/writing parts = atomic code
- When a loop performs a conditional check with a shared variable (i.e., flag).
  - Surround reading part (i.e., conditional block) and writing part (i.e., flag-flipping part) with lock() and unlock()
    - Reading/writing parts = atomic code
  - Try NOT to surround the entire loop with with lock() and unlock()!

Treating the Entire Loop as Atomic Code Does NOT Enjoy Concurrency

- Try NOT to surround the entire loop with lock() and unlock() as often as possible

```
– lock.lock();
  while(!done){  // reading part
      doThisTask();
      doThatTask();
      if(a condition is satisfied) break;
  }
  lock.unlock();

– lock.lock();
  done = true;  // writing part
  lock.unlock();
```

  - Even if the main thread acquires the lock earlier than the 2nd thread…
    - The main thread has a chance to quit the loop.
    - The 2nd thread has a chance to flip the flag.
    - However, the flip may occur a lot later than the 2nd thread wants.

Treating the Entire Loop as Atomic Code May Result in a Deadlock

- Should NOT surround the entire loop with lock() and unlock() in most cases

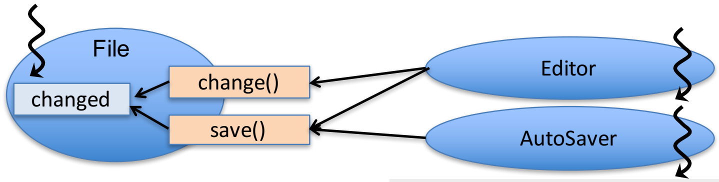
```
– lock.lock();
  while(!done){   // reading part
      System.out.println("#");
  }
  lock.unlock();

– lock.lock();
  done = true;   // writing part
  lock.unlock();
```

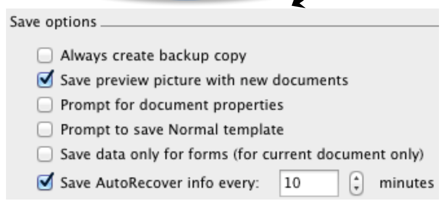
  - If the main thread acquires the lock earlier than the 2nd thread…
    - The main thread prints out #s forever.
    - The 2nd thread cannot flip the flag forever (deadlock!)
  - If the 2nd thread acquires the lock earlier than the main thread…
    - The 2nd thread flips the flag immediately.
    - The main thread print out no #s.

HW 9 [Optional]

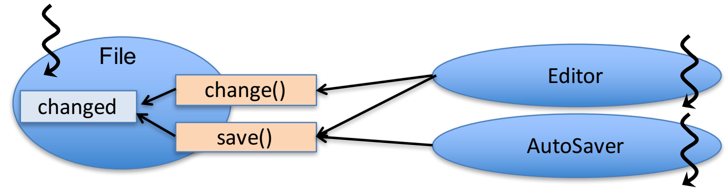
- Revise ThreadUnsafeFlag.java to be thread-safe.
  - Use ReentrantLock
  - Use balking
  - Use try-finally blocks.
    - Call unlock() in a finally block. Always do this in all subsequent HWs.

Exercise: Concurrent Access to a File



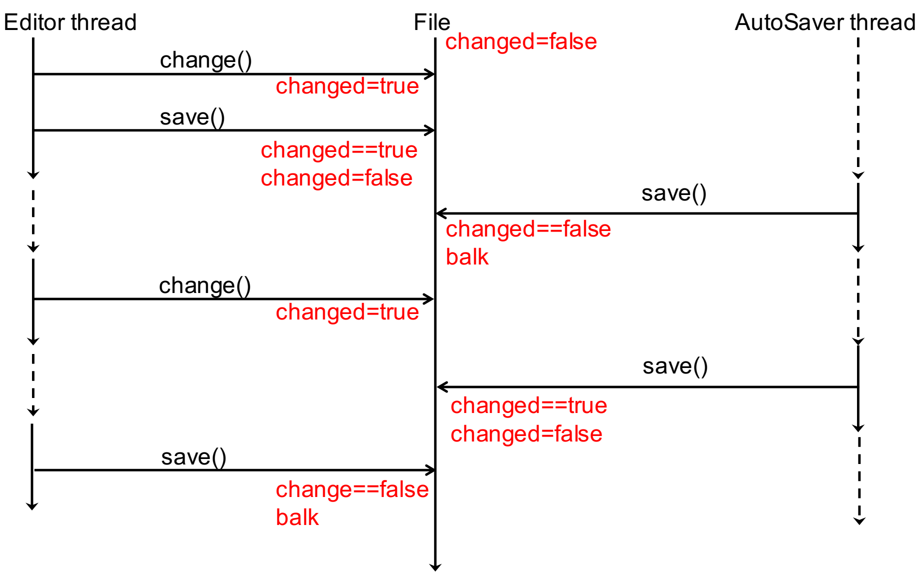


- Imagine word processing software.
- Assume two threads in it.
  - One for editing a file (Editor)
  - One for automatically saving a file (AutoSaver)
- Threads access a file concurrently.



- File
  - Has the variable "changed."
    - Initialized to be false.
  - change()
    - Changes the file's content.
    - Assigns true to the variable "changed."
  - save()
    - if (changed==false) return; // balking
    - if changed==true, print out some message (e.g., time stamp, etc.)
    - assigns false to the variable "changed."
- Editor (a Runnable) repeats:
  - Calls change() and save()
  - Sleeps for a second.
- AutoSaver (a Runnable) repeats:
  - Calls save() and
  - Sleeps for two seconds.

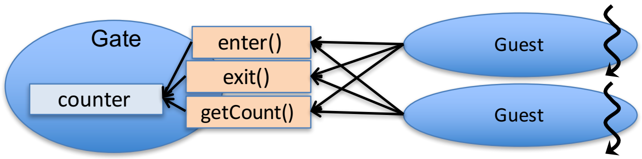
Desirable Result



HW 10

- Race conditions occur if you do not use a lock.
- Explain a potential race condition with a diagram like in the previous slide.
- Submit thread-safe code.
  - Define a lock in File. Use the lock in change() and save().
  - Create two extra threads.
  - Have them acquire and release the lock in change() and save().
  - Use balking
  - Use try-finally blocks
    - Call unlock in a finally block. <u>Always do this in all subsequent HWs.</u>

Exercise



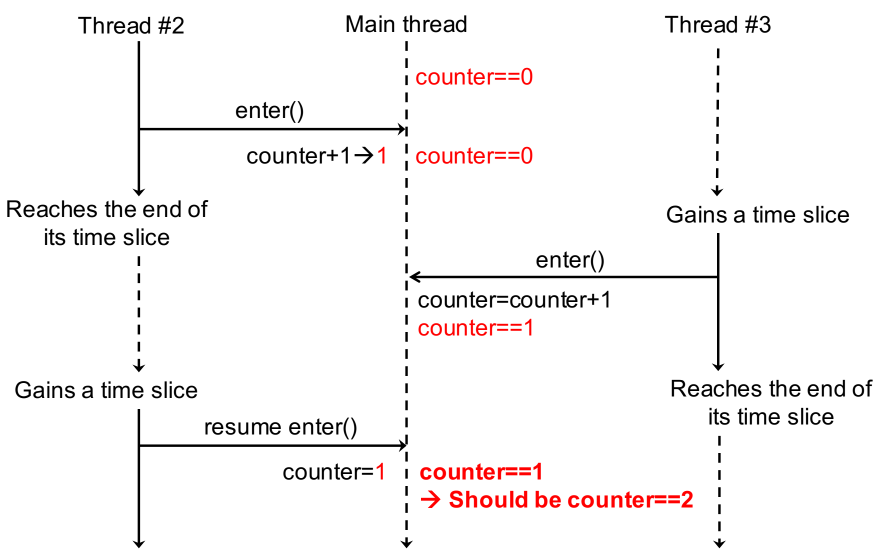

```
class Gate{
  private int counter = 0;

  public void enter(){
    counter++;
  }

  public void exit(){
    counter--;
  }

  //Get the # of guests in the gate
  public int getCount(){
    return counter;
  }
}
```

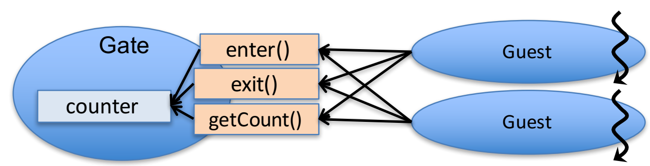
```
class Guest implements Runnable{
  private Gate gate;

  public Guest(......){
    gate = Gate.getInstance();
  }

  public void run(){
    gate.enter();
    gate.exit();
    gate.getCount();
  }
}
```

Not Thread Safe!

- `SecurityGate` is not thread-safe due to potential race conditions.
- `couter++` is a compound operation and not atomic.
  - Syntactic sugar for `counter = counter + 1;`
  - Requires more than one atomic operation (5 steps)
    - Called a *compound* operation.
    - A context switch can occur *across* different steps.
- The same goes to `counter--`.

A Potential Race Condition






```
class Gate{
  private int counter = 0;
  private ReentrantLock lock;

  public void enter(){
    lock.lock();
    counter++;
    lock.unlock();
  }

  public void exit(){
    lock.lock();
    counter--;
    lock.unlock();
  }

  public int getCount(){
    return counter;
  }}
```

```
class Guest implements Runnable{
  private SecurityGate gate;

  public UserAccess(......){
    gate = SecurityGate.getInstance();
  }

  public void run(){
      gate.enter();
      gate.exit();
      gate.getCount();
  }
}
```

An Alternative Solution: Use AtomicInteger

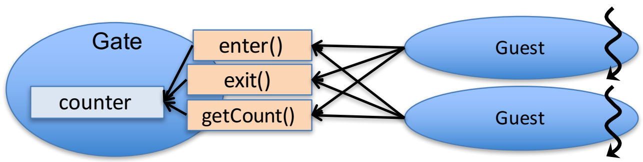
- Offers a series of methods to manipulate an integer value ***atomically***.
  - ```
    int i = ...
    i = i + 1;                    // Not thread safe
    ```
  - ```
    AtomicInteger atomicInt = new AtomicInteger(0);
    atomicInt.incrementAndGet();  // Thread safe
    ```
  - `addAndGet(int), decrementAndGet(), set(), get(), getAndSet(int), compareAndSet(int, int)...`
  - `accumulateAndGet(int, IntBinaryOperator), updateAndGet(IntUnaryOperator)`
    - `IntBinaryOperator, IntUnaryOperator`: functional interfaces
  - Many of the methods do not use locking; they are faster than lock-based code.

Be Careful!

- ```
  int i = ...
  i = i + 1;                              // Not thread safe
  ```
- ```
  AtomicInteger atomicInt = new AtomicInteger(0);
  atomicInt.incrementAndGet();            // Thread safe
  ```
- ```
  AtomicInteger atomicInt = new AtomicInteger(0);
  atomicInt.incrementAndGet();            // Thread safe
  ```
  - This is atomic.
- ```
  AtomicInteger atomicInt = new AtomicInteger(0);
  int i = atomicInt.incrementAndGet(); // NOT thread safe
  ```
  - This is a compound operation.

java.util.concurrent.atomic Package

- AtomicBoolean
- AtomicInteger
- AtomicLong
- AtomicReference<V>
- AtomicIntegerArray
- AtomicLongArray
- AtomicReferenceArray<E>
- ...




- ```
  class Gate{
   private int counter = 0;
   private ReentrantLock lock;

   public void enter(){
     lock.lock();
     counter++;
     lock.unlock();
   }

   public void exit(){
     lock.lock();
     counter--;
     lock.unlock();
   }

   public int getCount(){
     return counter;
   }}
  ```
- ```
  class Guest implements Runnable{
   private SecurityGate gate;

   public UserAccess(......){
    gate = SecurityGate.getInstance();
   }

   public void run(){
       gate.enter();
       gate.exit();
       gate.getCount();
   }
  }
  ```

HW 11

- Submit thread-safe code.
  – Do not use ReentrantLock.
  – Use AtomicInteger.

Regular and Static Locks

```
public class Foo{
    ReentrantLock  lock  = new ReentrantLock();
    static ReentrantLock  sLock  = new ReentrantLock();  }
```

- A regular lock is created and used on an *instance-by-instance* basis.
  – Different instances of Foo have different locks (i.e. different instances of ReentrantLock).
- A static lock is created and used on a *per-class* basis.
  – All instances of Foo share a single lock (“sLock”).

Exercise (1)

```
Public class Foo{
 private ReentrantLock lock = new ReentrantLock();
 private static ReentrantLock sLock = new ReentrantLock();
 public        void a(){…}
 public        void b(){…}
 public        void syncA(){lock.lock(); … lock.unlock();}
 public        void syncB(){lock.lock(); … lock.unlock();}
 public static void sA(){…}
 public static void sB(){…}
 public static void sSyncA(){sLock.lock(); … sLock.unlock();}
 public static void sSyncB(){sLock.lock(); … sLock.unlock();} }
```

- `x = new Foo(); y = new Foo();`
- Two threads call…
  – x.a() and x.a(): no synchronization b/w the two threads
  – x.a() and x.b(): no synchronization
  – x.a() and x.syncA(): no synchronization
  – x.syncA() and x.syncA(): Synchronization
  – x.syncA() and x.syncB(): Synchronization
  – x.syncA() and y.syncA(): No synchronization
  – x.syncA() and y.syncB(): No synchronization

Exercise (2)

```
Public class Foo{
 private ReentrantLock lock = new ReentrantLock();
 private static ReentrantLock sLock = new ReentrantLock();
 public        void a(){…}
 public        void b(){…}
 public        void syncA(){lock.lock(); … lock.unlock();}
 public        void syncB(){lock.lock(); … lock.unlock();}
 public static void sA(){…}
 public static void sB(){…}
 public static void sSyncA(){sLock.lock(); … sLock.unlock();}
 public static void sSyncB(){sLock.lock(); … sLock.unlock();} }
```

- `x = new Foo(); y = new Foo();`
- Two threads call…
  – x.a() and Foo.sA(): No synchronization b/w the two threads
  – x.syncA() and Foo.sA(): No synchronization
  – Foo.sA() and Foo.sA(): No synchronization
  – Foo.sA() and Foo.sB(): No synchronization
  – Foo.sSyncA() and Foo.sSyncA(): Synchronization
  – Foo.sSyncA() and Foo.sSyncB(): Synchronization
  – x.sSyncA() and y.sSynchB(): Synchronization
    - This is not grammatically wrong, but write Foo.sSyncA() instead of x.sSyncA()

Thread.sleep()

- The main thread runs the following code:
  - ```
    Thread t = new Thread( new FooRunnable() );
    t.start();
    try{
        t.sleep(1000);
    }catch(InterruptedException e){...}
    ```
- It looks like an extra thread (*t*) will sleep.
- However, the main thread will actually sleep
  - because sleep() is a ***static method*** of Thread.
    - Thread.sleep(): Causes the *currently executing thread* to sleep (temporarily cease execution) for the specified number of milliseconds
- DO NOT write t.sleep(…). It's misleading and error-prone.
- ALWAYS WRITE Thread.sleep(…).

Concurrent Singleton Design Pattern

- Guarantee that a class has only one instance.

```
public class Singleton{
 private Singleton(){};
 private static Singleton instance = null;
 private static ReentrantLock lock = new ReentrantLock();

 // Factory method to create or return a singleton instance
 public static Singleton getInstance(){
    lock.lock();
    if(instance==null)
       instance = new Singleton();
    lock.unlock();
    return instance;
 }
}
```


HW 12

```
public class Singleton{
 private Singleton(){};
 private         Singleton instance = null;
 private static ReentrantLock lock = new ReentrantLock();
 public static Singleton getInstance(){
     lock.lock();
     if(instance==null)
         instance = new Singleton ();
     lock.unlock();
     return instance; } }
```

- If you remove the above three red lines, this code becomes thread unsafe. (Race conditions can occur.) Explain a potential race condition with a diagram like in a previous slide.
- Complete the singleton class to be thread-safe.
- main()
  - Create multiple threads and have them call Singleton.getInstance().
  - Make sure that only one instance is created with System.out.println(Singleton.getInstance());

Where did the Synchronized Methods go?

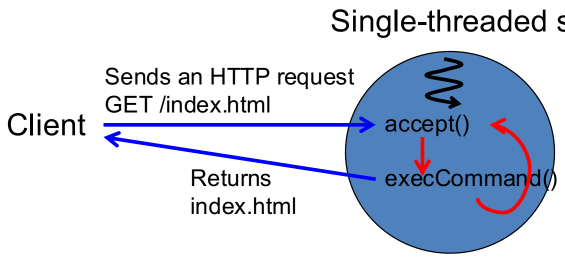
- Java still has the synchronized keyword.
  - `public synchronized void foo(){}`
    - Implicit locking. (The entire method body is atomic code.)
    - The scope of locking is always "per-method."
  - ```
    Public void foo(){
           // non-atomic code here
           synchronized(this){
                  // atomic code here
           }
           // non-atomic code here }
    ```
    - Threads try to acquire an implicit lock "this" maintains.
      - Instance-by-instance locking
    - Code looks tricky/dirty to use multiple locks in a single class.

- Explicit locking
  - ```
    ReentrantLock  aLock = new ReentrantLock()
    public void foo(){
        aLock.lock();
        // atomic code
        aLock.unlock(); }
    ```
  - Arbitrary locking scope.
  - Clean code even if a class uses multiple locks.
  - Extra functionalities
    - e.g., getQueueLength(): returns the # of waiting threads.
    - tryLock(): acquires a lock only if it is not held by another thread.
  - The catch is… it's VERY easy to forget calling unlock().
    - Must call unlock() in a finally clause.

- Implicit locking with the "synchronized" keyword
  - A thread can call notify() and notifyAll() even if it has not acquired a lock.
    - An IllegalMonitorStateException is thrown.
- Explicit locking
  - This error/bug never occurs.
    - ```
      ReentrantLock lock = new ReentrantLock();
      Condition cond = lock.newCondition();
      lock.lock();
      …
      cond.SignalAll();
      ```

Exercise: Access Counter for a Web Server

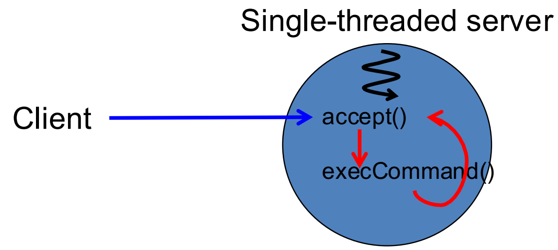
- Suppose you implement your own web server.
  - Receives a request that a client (browser) transmits to ask for an HTML file.
  - Returns the requested file to the client.
- What if the server receives multiple requests from multiple clients at the same time?
  - If the server is single-threaded, it processes requests *sequentially*.

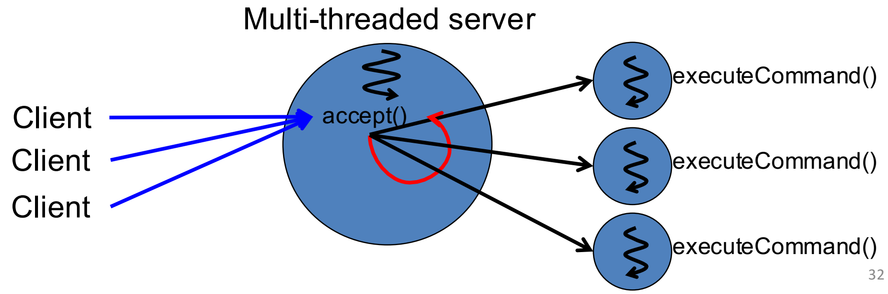


accept() waits/blocks for an incoming request. It returns once a request arrives at the server.

execCommand() parses the request and returns a requested file.

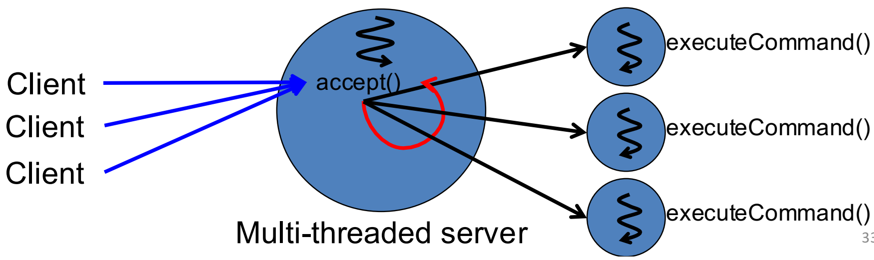
Concurrent (Multi-threaded) Web Server





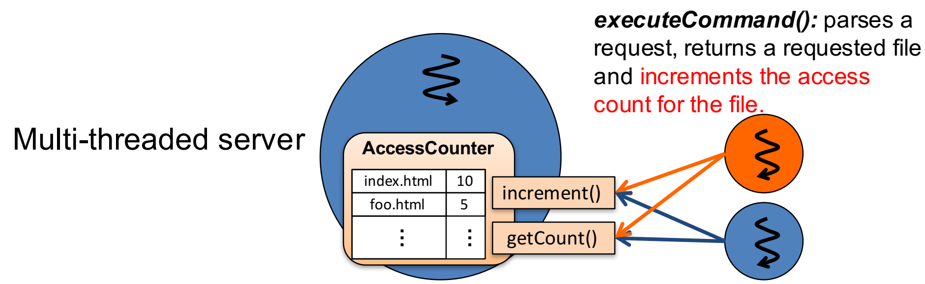
Thread-per-request Concurrency

- Thread-per-request
  - Once the web server receives a request from a client, it creates a new thread.
    - The thread parses the incoming request and returns a requested file.
    - The thread terminates once the requested file is sent out to the client.



Access Counter in a Concurrent Web Server

- `AccessCounter`
  - Maintains a map that pairs a relative file path and access count.
    - Assume `java.util.HashMap`
  - `increment()`
    - accepts a file path and increments the file's access count.
  - `getCount()`
    - accepts a file path and returns the file's access count.



Thread-safe Access Counter

- HashMap is NOT thread-safe.
  - All of its methods do not use a lock.
    - put(), putIfAbsent(), replace(), etc.
  - Race conditions can occur in those methods.

- Race conditions can occur in increment() and getCount() as well.
  - increment()
    - if(the path of a requested file is in AC){
      increment the access count for that path. }
      else{
      add that path the access count of 1 to AC. }
  - getCount()
    - if(the path of a requested file is in AC){
      get the access count for that path and return it. }
      else{
      return 0. }

HW 13

- Explain how a potential race condition occurs in increment() and getCount().
  - Draw sequence diagrams.

- Implement thread-safe AccessCounter
  - Define a `HashMap<java.nio.Path,Integer>`
    - c.f. Lec note #1 about `java.nio.Path`
  - Use a lock in increment() and getCount()

- Place some text files
  - AccessCounter
    RequestHandler (implementing Runnable)
      file_root
        a.html
        b.html
        …

- RequestHandler: A Runnable class
  - run(): Picks up one of the files, and calls increment() and getCount() for that file.

- main()
  - Creates and starts multiple threads (e.g., 10+ threads) to access AccessCounter concurrently.

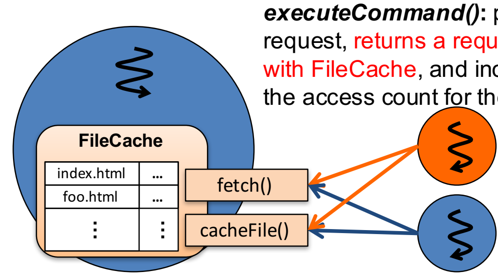
Exercise: File Caching in a Web Server

- File caching
  - Keep frequently-accessed files in the memory space rather than external/peripheral storages such as HDDs.
    - Obtain a requested file from an external storage for the first time (i.e., for the first request)
    - Keep (or cache) the file in the memory space
    - Use the cached file from the next time
      - Skip the overhead to access an external storage
  - Can improve the performance (response time and throughput) of your web server.

File Caching in a Concurrent Web Server

- Keep assuming a thread-per-request concurrency.
- **`FileCache`**
  - Maintains a map that pairs a relative file path and string data of the file.
    - Assume **`java.util.HashMap`**
  - **`fetch()`**
    - accepts a file path and gets the content of the requested file from the HashMap.
  - **`cacheFile()`**
    - accepts a file path and its content to the HashMap.

Multi-threaded server



executeCommand(): parses a request, returns a requested file with FileCache, and increments the access count for the file.

- Define an abstract class: **`FileCache`**
  - **`public String fetch(targetFile)`**
    - targetFile: a relative file path
      - String or java.nio.Path
    - Returns a requested file's content
    - if (targetFile is cached){
        return targetFile's content (as String);
      else
        return cacheFile(targetFile); }
  - **`private String cacheFile(…)`**
    - If(cache is full){
        replace(targetFile)
      }else{
        cache targetFile;
        return its content; }
- Define multiple classes that extend **`FileCache`**
  - Each class implements replace()

- FileCache maintains a *fixed-sized* cache.
  - Maintains the max number of path-content pairs.
  - Performs *cache replacement* when the number of path-content pairs exceeds the threshold.
  - Implements each cache replacement policy.
    - Least Recently Used (LRU)
      - Replaces the least recently requested file with a new file.
      - Keep the timestamp for each file in the cache.
    - Least Frequently Used (LFU)
      - Replaces the least frequently requested file with a new file.
      - Uses AccessCounter

HW 14

- Implement FileCache in a thread-safe way
  - Use **`String`** or **`java.nio.Path`** to represent a file path
  - Use a lock in fetch(), cacheFile() and replace(). Assume nested locking.
- Place some text files
  - FileCache (abstract class)
    FileCacheLRU (extending FileCache)
    FileCacheLFU (extending FileCache)
    AccessCounter
    RequestHandler (implementing Runnable)
      file_root
        a.html
        b.html
        …
- RequestHandler.run()
  - Picks up one of the files and calls fetch() for that file.
  - Calls AccessCounter's increment() and getCount() as well.
- main()
  - Creates and starts multiple threads (e.g., 10+ threads) to access FileCache and AccessCounter concurrently.