Revisiting Explicit Thread Termination

Explicit Thread Termination

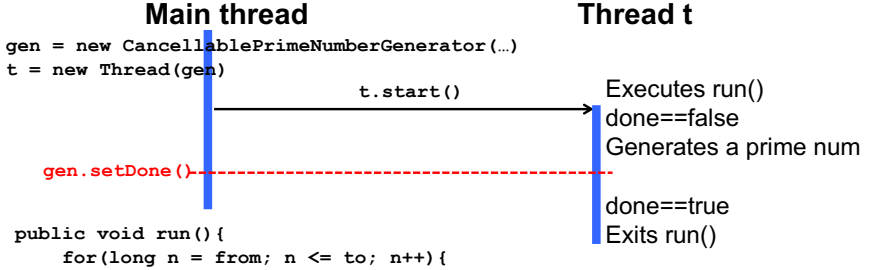
- 2 approaches
  - Flag-based
  - Interruption-based

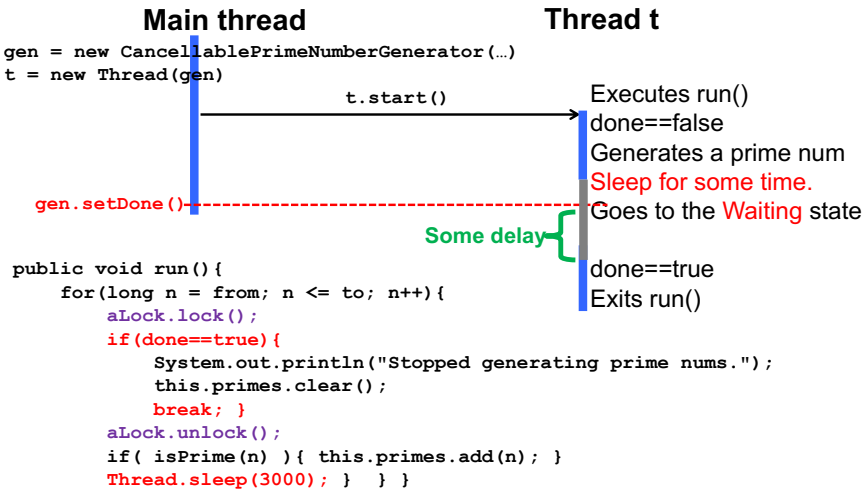
Explicit Thread Termination

- Which way to go?
  - Flag-based
    - Pros:
      - Uses 1 lock (faster)
    - Cons:
      - Need to define and maintain a flag by yourself.
      - Program responsiveness may be lower.
        - » if a flag-flipping (e.g. done==false → true) happens when a thread to be terminated is in the Waiting or Blocked state.
  - Interruption-based
    - Pros
      - No need to define and maintain a flag
      - Higher program responsiveness
        - » interrupt() can immediately wake up a thread that is in the Waiting or Blocked state
    - Cons
      - Uses 2 locks (slower)

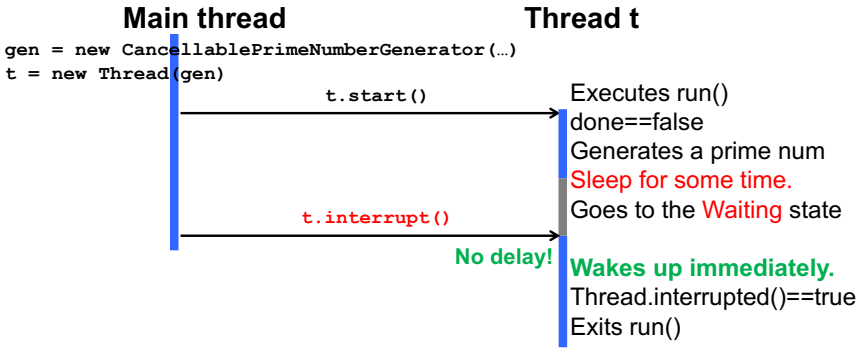
When a Flag-flipping Occurs…

- If a thread to be terminated (***t***) is in the Runnable state when a flag-flipping occurs…





Thread Interruption can Improve Program Responsiveness



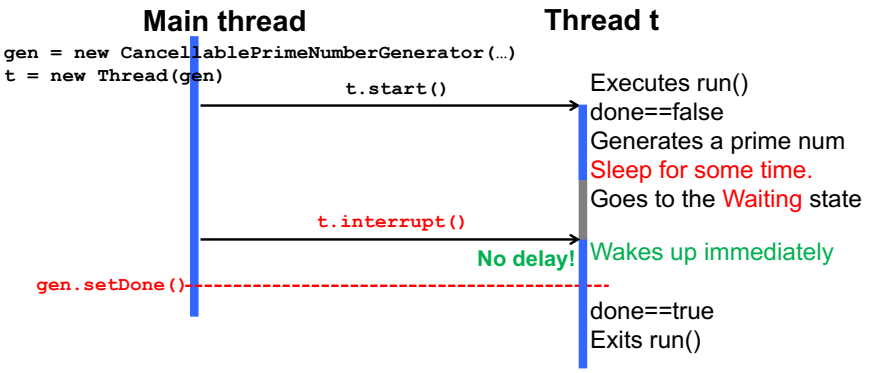
Hybridization of the Two Approaches?

- Can we implement a responsive thread termination that uses only 1 lock?

2-Step Thread Termination ("Graceful" Thread Termination)

2-Step Thread Termination

- Primarily takes the flag-based approach.
  - A thread to be terminated repeatedly checks a flag.
- Let the "terminator" thread call interrupt() before flag-flipping (e.g. calling setDone())



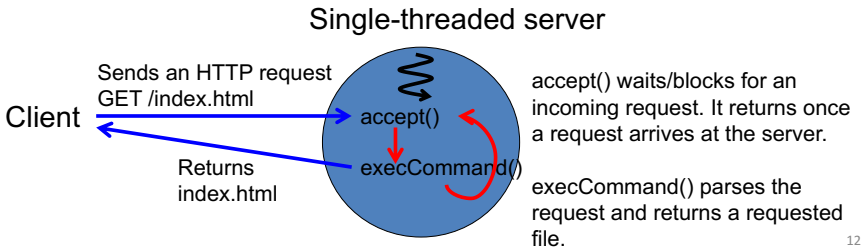

```
public void run(){
    for(long n = from; n <= to; n++){
        aLock.lock();
        if(done==true){
            System.out.println("Stopped generating prime nums.");
            this.primes.clear();
            break; }
        aLock.unlock();
        if( isPrime(n) ){ this.primes.add(n); }
        Thread.sleep(3000); }  } }
```

HW 11

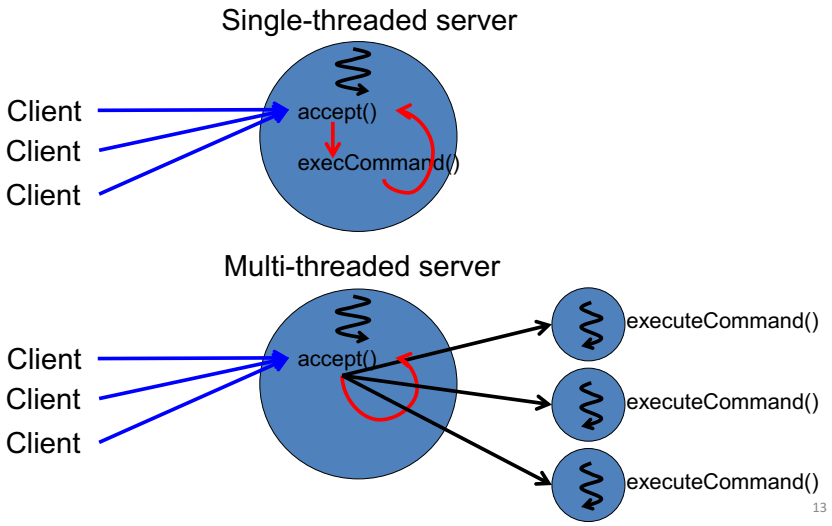
- Revise a thread-safe version of CancellablePrimeNumberGenerator to perform 2-step thread termination.
- Deadline: April 17 (Tue) midnight

Exercise: Access Counter for a Web Server

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

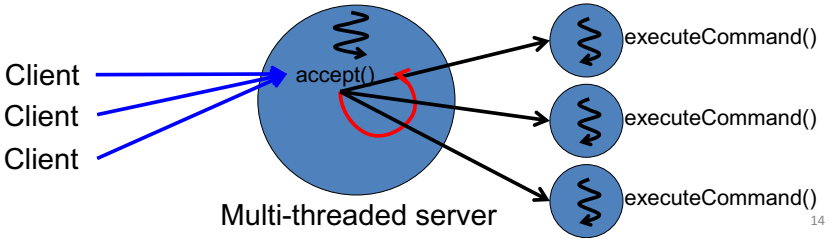


Concurrent (Multi-threaded) Web Server



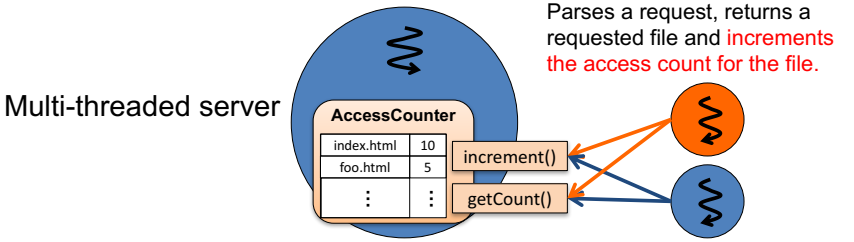
Thread-per-request Concurrency

- 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 returned 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.



Concurrent Access Counter

- HashMap is NOT thread-safe.
  - All of its methods never do locking.
    - 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 and 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 12

- Implement AccessCounter in a thread-safe manner.
  - Define a `HashMap<java.nio.Path, Integer>`
    - c.f. Lec note #1 about `java.nio.Path`
  - Define a lock and use a lock in increment() and getCount()
- Place some text files
  - AccessCounter.java
    RequestHandler.java (implementing Runnable)
    file_root
      a.html
      b.html
      …
- RequestHandler: A Runnable class
  - run(): Picks up one of the files at random, calls increment() and getCount() for that file, and sleep for a few seconds
- main()
  - Creates and starts 10+ threads to access AccessCounter concurrently.
- Implement 2-step thread termination.
  - Have the main thread terminate those 10+ threads in 2 steps.

- Deadline: April 17 (Tue) midnight

Volatile Variables in Java

What is the "volatile" Keyword?

- You must do locking to share variables among threads.
- You can skip locking to share *volatile* variables among threads in some cases.
  - No locking means…
    - No worry about concurrency problems
      - Particularly, race conditions
    - Less overhead (better performance)
      - Locking consumes some time and resources.

Syntactic Difference

- Without "volatile"

```
boolean done = false;
ReentrantLock lock = …;

public void setDone(){
 lock.lock();
 done = true;
 lock.unlock();
}

public void run(){
  while( true ){
     lock.lock();
     if( done ) break;
     counter++;
     lock.unlock(); }
}
```

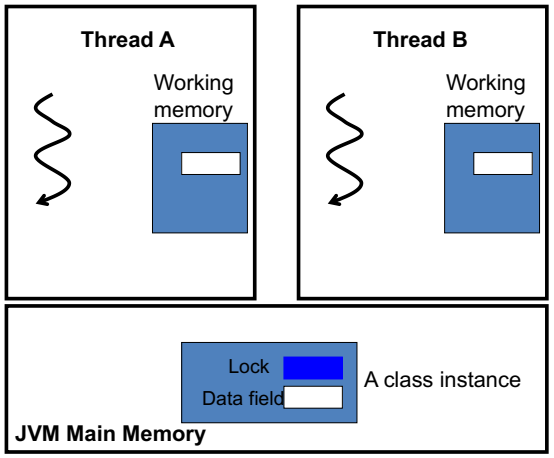
- With "volatile"

```
volatile boolean done = false;

public void setDone(){
  done = true;
}

public void run(){
  while( true )
    if( done ) break;
    counter++;
}
```


JVM Memory Model (J2SE 5.0-)



Main memory

Shared by all threads

Working (local) memory

Per-thread space

Not accessible/visible by the other threads

A local copy of each data field

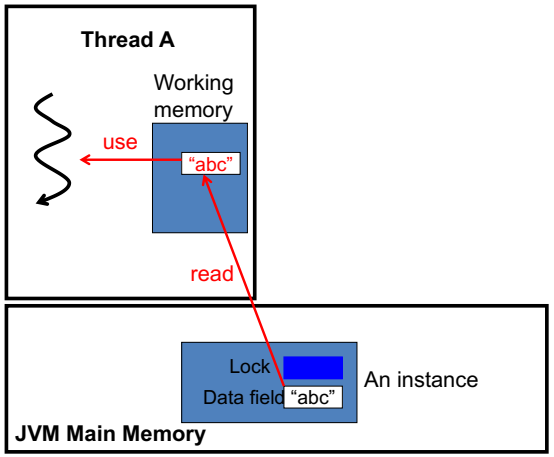
Local variables

JVM Actions

- JVM implements a set of *JVM actions* to manage the main memory space and local memory spaces.
- JVM actions (bytecode instructions)
  - *read, write, use, assign*
  - *lock, unlock*
  - All atomic

Read Operation (Single Threaded)

```
System.out.println(variable);
```



When thread A reads a value for the first time…

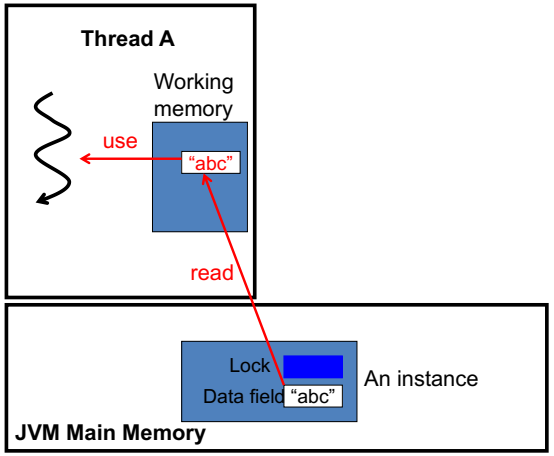
(1) Read the value from the main memory

(2) Store the value in the local working memory.

Always Read-Use for the first read operation.

Read Operation (Single Threaded)

```
if(variable.equals(...))
```



When thread A reads a value for the first time…

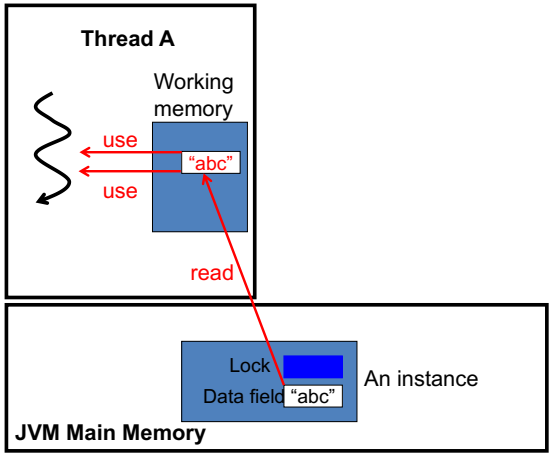
(1) Read the value from the main memory

(2) Store the value in the local working memory.

Always Read-Use for the first read operation.

Read Operation (Single Threaded)

```
if(variable.equals(...)){
  System.out.println(variable); }
```



When thread A reads a value for the first time…

(1) Read the value from the main memory

(2) Store the value in the local working memory.

Always Read-Use for the first read operation.

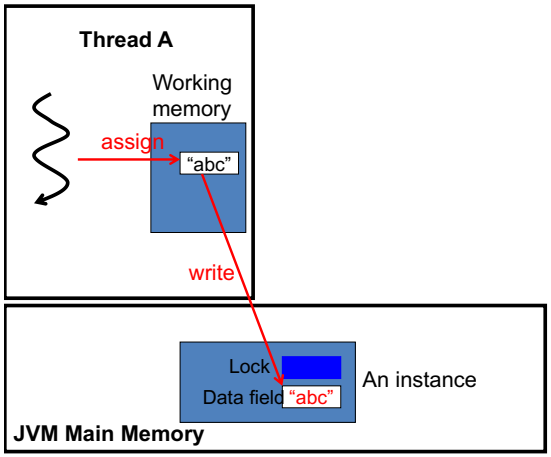
When thread A accesses the same variable later…

Use OR Read-Use

It's up to JVM implementations, but often, Use only.

Write Operation (Single Threaded)

```
variable = "abc";
```



When thread A assigns a value to a data field…

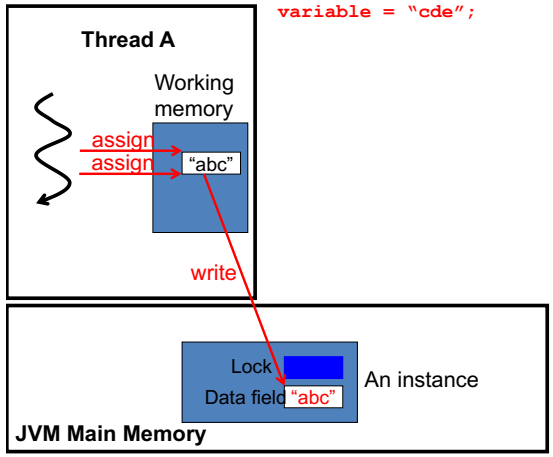
(1) Assign the value to a data field in the local memory.

(2) Copy it to the main mem.

Always Assign-Write for the first assignment operation.

Write Operation (Single Threaded)

```
variable = "abc";
variable = "cde";
```



When thread A assigns a value to a data field…

(1) Assign the value to a data field in the local memory.

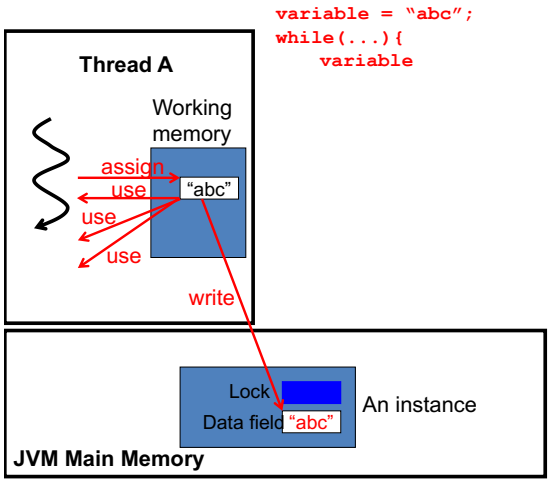
(2) Copy it to the main mem.

Always Assign-Write for the first assignment operation.

It is not predictable when (how soon) "write" occurs. Maybe immediately after "assign," but some interval may exist.

If thread A repeatedly assigns different values to the the same data field, it makes sense to do: Assign, Assign, … , Write

Write Operation (Single Threaded)



When thread A assigns a value to a data field…

(1) Assign the value to a data field in the local memory.

(2) Copy it to the main mem.

Always Assign-Write for the first assignment operation.

It is not predictable when (how soon) "write" occurs. Maybe immediately after "assign," but some interval may exist.

If thread A repeatedly assigns different values to the the same data field, it makes sense to do: Assign, Assign, … , Write

If thread A often read the value, it makes sense to do: Assign, Use, Use, Use, … , Write

JVM Actions

- JVM implements a set of *JVM actions* to manage the main memory space and local memory spaces.
- JVM actions (bytecode instructions)
  - *read, write, use, assign*
  - *lock, unlock*
  - All atomic

Thread and Memory Sync

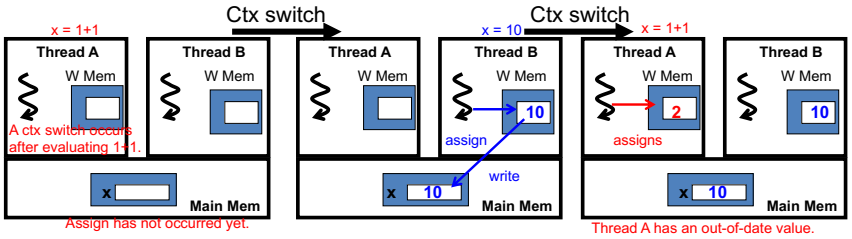
- When a thread runs atomic code…
  - public void setDone(){
    lock.lock()
    // atomic code;
    lock.unlock(); }
- JVM does two things:
  - *Thread synchronization*
    - Only one thread enters and executes atomic code at a time.
      - All the other threads are blocked.
  - *Memory synchronization*
    - Synchronize the most up-to-date value in between a local memory and the main memory.

Race Condition

- A race condition can occur due to
  - Failure of thread synchronization and/or
  - Failure of memory synchronization
    - Inconsistency between data in working and main memories.

Race Condition

- A race condition can occur due to
  - Failure of thread synchronization and/or
  - Failure of memory synchronization
    - Inconsistency between data in working and main memories.

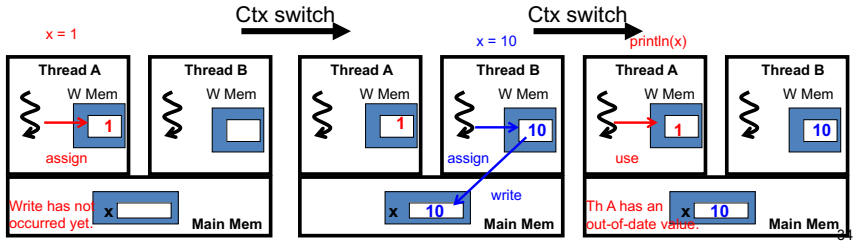


A race condition can occur due to a failure of thread synchronization.

Race Condition

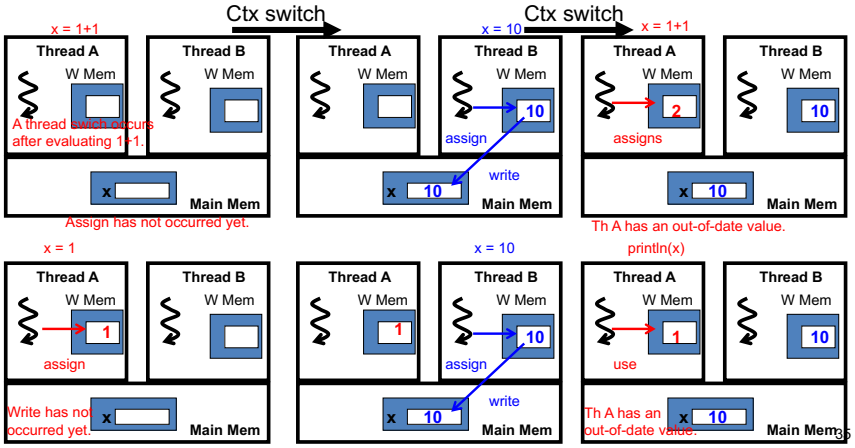
- A race condition can occur due to
  - Failure of thread synchronization and/or
  - Failure of memory synchronization
    - Inconsistency between data in working and main memories.

A race condition can occur due to a failure of memory synchronization. Threads synchronized in this case (by chance).



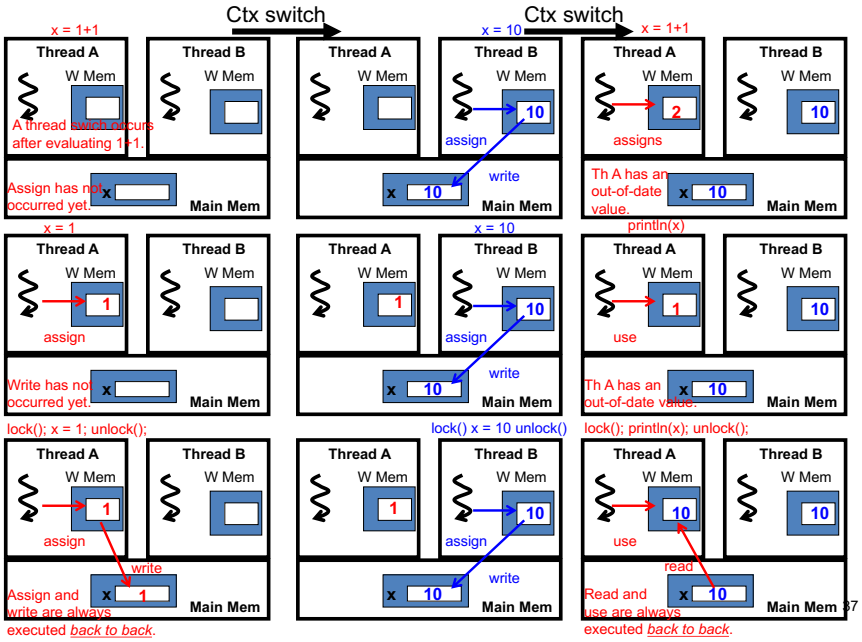
Race Condition

- A race condition can occur due to
  - Failure of thread synchronization
  - Failure of memory synchronization
    - Inconsistency between data in working and main memories.



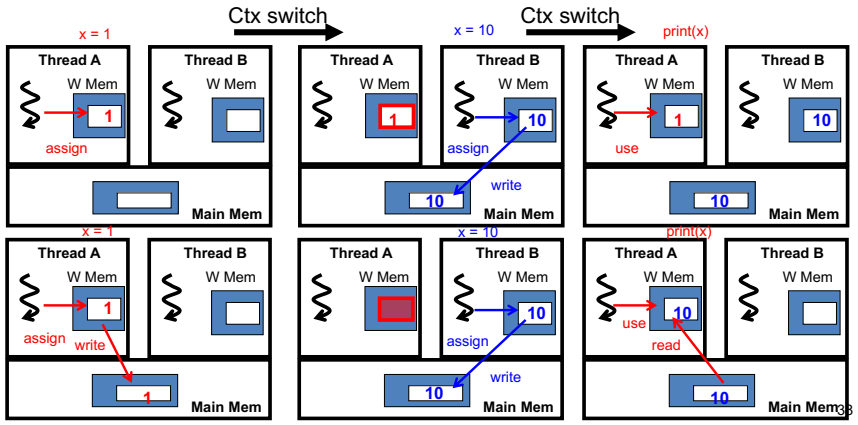
Race Conditions and Locking

- You need both thread and memory synchronization to prevent race conditions.
- Locking does both.
  - Thread synchronization
    - Only one thread enters and executes atomic code at a time.
      - All the other threads are blocked.
  - Memory synchronization
    - Synchronize the most up-to-date value in between a local memory and the main memory.
      - Destroy working memory upon entering atomic code
      - Flush working memory to the main memory upon existing atomic code

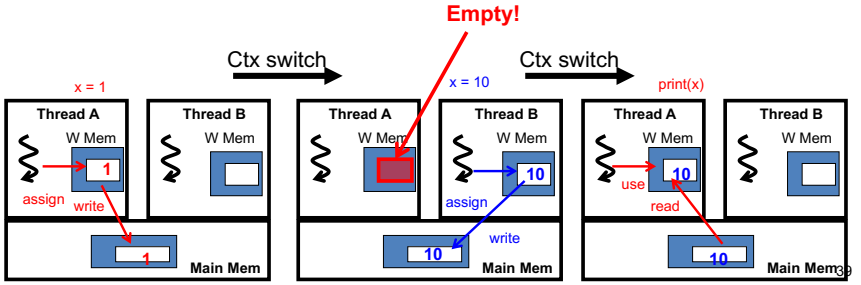


Volatile Variables

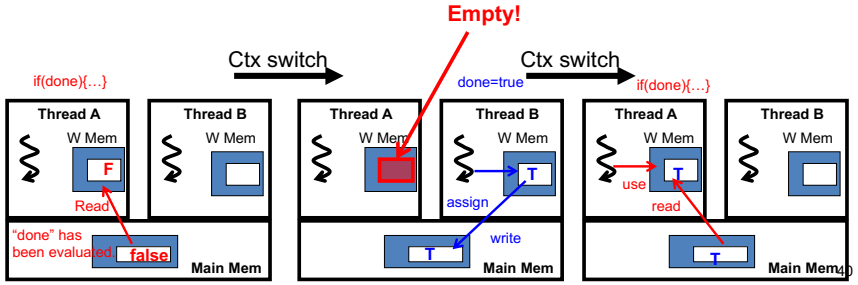
- When a thread uses a a volatile variable…
  - Memory synchronization is guaranteed.



- Write always follows Assign immediately.
  - They are always paired.
- Read always occur before Use.
  - They are always paired.
- A volatile variable's value is *NOT persistent (i.e., volatile)* across context switches.



- Write always follows Assign immediately.
  - They are always paired.
- Read always occur before Use.
  - They are always paired.
- A volatile variable's value is *NOT persistent (i.e., volatile)* across context switches.



- No need to use locking to perform memory synchronization for a volatile variable.
  - No locking → less overhead
- A volatile variable NEVER synchronizes threads that access the variable.
  - "Volatile" is not a thread sync tool. It's for a memory sync tool.
- ***The volatile keyword works when you don't need to synchronize threads but need to synchronize memory.***

When to use Volatile Variables?

- When a value to write into a shared variable does not depend on the current value.
- Latch
  - A data structure that performs a single type of state changes
    - e.g. False → True
  - Often used to terminate threads.
    - c.f. "done" variable in prior examples

An Example Latch

```
volatile boolean done = false;

public void setDone(){
  done = true;}

public void run(){
  while(true)
    if(done) break;
    counter++;
}
```

- The state of "done" always changes in a unidirectional way: false → true
  - "true → false" never occur.


```
volatile boolean done = false;

public void setDone(){
  done = true;}

public void run(){
  while(true)
    if(done) break;
    counter++; }
```

- No need to surround the if statement with lock() and unlock().
- Thread sync is not performed.
  - A context switch can occur in between
    - Evaluating the "done" variable and
    - Applying the current value in "done" to the if statement
- Memory sync is performed.
  - The most up-to-date value of "done" is applied to the if statement.

```
volatile boolean done = false;

public void setDone(){
  done = true;}  // NO NEED TO SURROUND THIS WITH LOCK() and UNLOCK()

public void run(){
  while(true)
    if(done) break;
    counter++; }
```

- Thread sync is not performed.
  - A context switch can occur in between evaluating the value of "true" and assigning it "done."
  - All threads will assign "true" to "done."
    - No other possible state changes.
  - Writer threads do not generate race conditions.
- Memory sync is performed.
  - The value of "true" must be copied to the main memory once the assignment ("done=true") is completed.

Syntactic Difference

- Without "volatile"

```
boolean done = false;
ReentrantLock lock = …;

public void setDone(){
 lock.lock();
 done = true;
 lock.unlock();
}

public void run(){
  while( true ){
     lock.lock();
     if( done ) return;
     counter++;
     lock.unlock(); }
}
```

- With "volatile"

```
volatile boolean done = false;

public void setDone(){
  done = true;}

public void run(){
  while( true )
    if( done ) break;
    counter++;
}
```

- Or …

```
public void run(){
  while( !done )
    counter++;
}
```


Notes

- volatile int a = 0; a = 1;
  - The value of 1 is copied to the main memory right after a=1 is done in the local memory.
  - A context switch can occur in between evaluating the value of 1 and assigning it to the variable "a."
    - You can use "volatile" only when you are fine with that.
- int a = 0; a = 1;
  - The value of 1 may not be copied to the main memory right after a=1 is done in the local memory.
  - A context switch can occur in between completing a=1 in the local memory and copying it to the main memory.
- "Volatile" works for ***single*** write/read operations, which have no intermediate states.
  - volatile a;
    a = a + 1; // "volatile" does not work properly here because this is compound.
- Do not use volatile for arrays.

In Summary…

- Not a silver bullet
  - Not a general-purpose, widely-applicable threading tool
- Useful only in some specific cases
  - In practice, assume it is useful only for simplifying the implementation of a latch.
    - Use it only for implementing flag-based thread termination.

A Concurrency Bug in Jetty

- Jetty
  - An open source web implementation in Java
    - http://jetty.codehaus.org/jetty/
- A bug report (March 2010)
  - http://jira.codehaus.org/browse/JETTY-1187
  - ```
    // Jetty 7.1.0,
    // org.eclipse.jetty.io.nio,
    // SelectorManager.java, line 105
    private volatile int _set;
    public void register(SocketChannel channel, Object att){
        int s = _set++;
        ...
    }
    public void addChange(Object point){
        synchronized (_changes){
            ...
        }
    }
    ```
    ```