CMSC 433 – Programming Language Technologies and Paradigms

Thread Safety
Thread Safety

- Scheduler can interleaved or overlap threads arbitrarily
- Data can be shared by threads
- Can lead to interference
  - Storage corruption
  - Violation of representation invariant
  - Violation of a protocol (e.g., $A$ occurs before $B$)
public class Example extends Thread {
    private static int count = 0;  // shared state
    public void run() {
        int y = count;
        count = y + 1;
    }
    public static void main(String args[]) {  
        Thread t1 = new Example();
        Thread t2 = new Example();
        t1.start();
        t2.start();
    }
}
Example

```java
static int count = 0;  // Shared state

void t1.run() {
    int y = count;
    cnt = y + 1;
}

void t2.run() {
    int y = count;
    count = y + 1;
}
```

Start: both threads ready to run. Each will increment the global count.
static int count = 0;  

```java
    t1.run() {
        int y = count;  
        cnt = y + 1;
    }
```

```java
    t2.run() {
        int y = count;  
        count = y + 1;
    }
```

This example illustrates the concept of shared state in a multi-threaded environment.

**Shared state**

```
    count = 0
```

In the example, thread T1 executes, grabbing the global counter value into y. Then, T1 increments the counter and stores the new value back into the shared variable count.

```
    T1 executes, grabbing the global counter value into y.
```

This process highlights the potential for race conditions and the importance of proper synchronization techniques.
static int count = 0;  \textit{Shared state} \quad \text{count} = 1

t1.run() {
    int y = cnt; \quad y = 0
    count = y + 1;
}

t2.run() {
    int y = count; \quad T1 \text{ executes again, storing the} \text{ counter value}
    count = y + 1;
}
Example

```java
static int count = 0;

t1.run() {
    int y = count;  // y = 0
    count = y + 1;
}

t2.run() {
    int y = count;  // y = 1
    count = y + 1;
}
```

**Shared state**

```
count = 1
```

T1 finishes. T2 executes, grabbing the global counter value into y.
Example

```
static int count = 0;  // Shared state

    count = 2

    t1.run() {
        int y = count;  // y = 0
        count = y + 1;
    }

    t2.run() {
        int y = count;  // y = 1
        count = y + 1;
    }

T2 executes, storing the incremented count value.
```
But When I Run it Again?
Example

static int count = 0;  

Start: both threads ready to run. Each will increment the global count.

t1.run() {
    int y = count;
    cnt = y + 1;
}

t2.run() {
    int y = count;
    count = y + 1;
}
static int count = 0;  \hspace{1cm} \textit{Shared state} \hspace{1cm} count = 0

t1.run() {
    int y = count; \hspace{1cm} y = 0
    count = y + 1;
}

t2.run() {
    int y = count;
    count = y + 1;
}

\textit{T1 executes, grabbing the global counter value into y.}
Example

```java
static int count = 0;  // Shared state
t1.run() {
    int y = count;      // y = 0
    count = y + 1;
}
t2.run() {
    int y = count;      // y = 0
    count = y + 1;
}
```

T1 is pre-empted. T2 executes, grabbing the global counter value into y.
static int count = 0;                   \textit{Shared state} \quad \text{count} = 1

\begin{verbatim}
t1.run() {
    int y = count;  \quad y = 0
    count = y + 1;
}

\begin{verbatim}
t2.run() {
    int y = count;  \quad y = 0
    count = y + 1;
}
\end{verbatim}
\end{verbatim}

\textit{T2 executes, storing the incremented cnt value.}
Example

```java
static int count = 0;

t1.run() {
    int y = count;
    count = y + 1;
}

t2.run() {
    int y = count;
    count = y + 1;
}
```

*Shared state*

```
count = 1
```

T2 completes.  T1 executes again, storing the old counter value (1) rather than the new one (2)!
What Happened?

- The code read the counter value & then increments that value by one
- In the first example, \( t_1 \) was preempted after it read the counter but before it stored the new value.
- When \( t_1 \) resumed, it updated a stale value
- This is an example of a data race
Data Races & Race Conditions

- A *data race* occurs when two concurrent threads access a shared variable
  - at least one access is a write and
  - the threads use no explicit mechanism to prevent the accesses from being simultaneous

- A *race condition* occurs when a program’s correctness unexpectedly depends on the ordering of events
• If instead of writing
  ```
  int y = cnt;
  cnt = y+1;
  ```
• We had written
  - `cnt++;
• Would the result be any different?
• Answer: NO!
  - `++` operation is just a syntactic shorthand for the two operations above
• If you run a program with a race condition, will you always get an unexpected result?
  – No! It depends on the scheduler
  – ...i.e., which JVM you’re running
  – ...and on the other threads/processes/etc. that are running on the same CPU

• Schedule-driven problems are hard to reproduce
Atomicity

• A particular way in which the execution of two threads is interleaved is called a schedule

• We want to prevent undesirable schedules such as those shown in the counting example
• One way to prevent undesirable schedules is to ensure that the code in the two threads is atomic
  – Operations A and B are atomic with respect to each other if, from the perspective of the thread executing A, when another thread executes B, either all of B has executed or none of it has.
  – An atomic operation is one that is atomic with respect to all operations, including itself, that operate on the same state.
Locks

• Commonly used to enforce atomicity
  – Descends from semaphore construct in OS research & design

• Only one thread can hold a lock
  – Other threads block until they can acquire it
  – The operation of acquiring a lock is atomic
    • Cannot have a race on lock operations themselves!

• In Java every Object has (can act as) a lock
  – Called an *intrinsic lock*
Synchronized Statement

• `synchronized (obj) { statements }`
  – Acquires *(locks)* the `obj` intrinsic lock before executing statements in block
  – Releases *(unlocks)* the lock when the statement block completes, whether due to a break, return, exception, etc.
Avoiding Interference: Synchronization

```java
public class Example extends Thread {
    private static int count = 0;
    static Object lock = new Object();
    public void run() {
        synchronized (lock) {
            int y = count;
            count = y + 1;
        }
    }
    ...
}
```

- **Lock**, for protecting the shared state
- **Acquires the lock**; Only succeeds if not held by another thread
- **Releases the lock**
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
```

Shared state  count = 0

T1 acquires the lock
Applying Synchronization

```java
int count = 0;
t1.run() {
    synchronized(lock) {
        int y = count;  // T1 reads count into y
        count = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
```

Shared state  count = 0

T1 reads count into y
Applying Synchronization

```java
t1.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
```

Shared state: count = 0

- **T1 is pre-empted.**
- **T2 attempts to acquire the lock but fails because it’s held by T1, so it blocks.**
Applying Synchronization

```java
int count = 0;
t1.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
```

**Shared state**

```java
Shared state count = 1
```

**T1 runs, assigning to count**

```
y = 0
```
Applying Synchronization

```java
int count = 0;
t1.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
```

Shared state

- count = 1

T1 releases the lock and terminates
Applying Synchronization

```java
int count = 0;
t1.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
```

*Shared state*  
*count = 1*

*T2 now can acquire the lock.*
int count = 0;
t1.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
Applying Synchronization

```java
int count = 0;
t1.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = count;
        count = y + 1;
    }
}
```

Shared state: count = 2

T2 assigns count, then releases the lock.
More on Locks

- Intrinsic locks are reentrant
  - The thread can reacquire the same lock many times
  - Lock is released when object unlocked the corresponding number of times

- No way to attempt to acquire an intrinsic lock
  - Either succeeds, or blocks the thread
  - Java 1.5 java.util.concurrent.locks package added separate locks with more operations (will discuss these later in the semester)
Synchronized Methods

- A method can be synchronized
  - Add `synchronized` modifier before return type
- Obtains the lock on object referenced by `this` before executing method
  - Releases lock when method completes
- For a `static synchronized` method
  - Locks the `Class` object for the class
    - Accessible directly, e.g. `Foo.class`
Synchronization Style

• Internal sync. (class is thread-safe)
  – Have a stateful object synchronize itself (e.g., with synchronized methods). Robust to threaded callers
  – E.g., class Math.Random

• External sync. (class is thread-compatible)
  – Have callers perform synchronization before calling the object
  – If they don’t, behavior may be unpredictable
public class State {
    private int count = 0;
    public int synchronized incCount(int x) {
        count += x;
    }
    public int synchronized getCount() { return count; }
}

public class MyThread extends Thread {
    State s;
    public MyThread(State s) { this.s = s; }
    public void run() {
        s.incCount(1);
    }
    public void main(String args[]) {
        State s = new State();
        MyThread thread1 = new MyThread(s);
        MyThread thread2 = new MyThread(s);
        thread1.start(); thread2.start();
    }
}
public class MyThread extends Thread {
    static List l = new ArrayList();
    String s; // set in constructor
    void add(String s) {
        synchronized (l) { l.add(s); }
    }
    boolean check(String s) {
        synchronized (l) {
            return l.contains(s);
        }
    }
    public void run() {
        if (!check(s)) add(s);
    }
    public void main(String args[]) {
        MyThread thread1 = new MyThread("hello");
        MyThread thread2 = new MyThread("hello");
        MyThread thread3 = new MyThread("goodbye");
        thread1.start(); thread2.start(); thread3.start();
    }
}

Synchronization occurs in the caller of ArrayList (which is MyThread), not ArrayList itself.
Lack of data races = atomicity?

• For the previous example:
  – Are there any data races?
  – Are there any race conditions?

• What will value will the static variable l have at the end of an execution?
Answer

• There are no data races
  – All accesses are synchronized

• There is a race condition
  – Race condition caused by a violation of atomicity.
  – We expect the output to be \{ “hello”, “goodbye” \}
  – But in fact it could also be \{ “hello”, “hello”,
    “goodbye” \}
• This is an example of a compound action
  – A sequence of operations that need to be atomic
• Common examples
  – Read-modify-write
  – Check-then-act
public class MyThread extends Thread {
    static List l = new ArrayList();
    String s;
    public void run() {
        synchronized (l) {
            if (!l.contains(s))
                l.add(s);
        }
    }
}

public void main(String args[]) {
    MyThread thread1 = new MyThread("hello");
    MyThread thread2 = new MyThread("hello");
    MyThread thread3 = new MyThread("goodbye");
    thread1.start(); thread2.start();
    thread3.start();
}
Is the String class thread-safe or thread-compatible?
- Fact: none of its methods are annotated with the keyword “synchronized”

It’s thread-safe, since callers don’t need to do additional synchronization

String objects are immutable, so they can be freely shared across threads
Concurrent Rule of Thumb

• Perform actions only when in consistent states
  – Don’t want one thread to access an object while another thread is modifying its internal state.

• This boils down to ensuring object invariants in the face of concurrent access
A Thread-Safe Multi-Threaded Logging Server

- **Logging server**
  - Accepts records from client
  - Creates a thread that writes the record to client-specific file

- **Organization**
  - **Utils**
    - DataRecord.java
    - LockingMsgHandler.java
  - **Client**
    - ClientSimulator.java
  - **Server**
    - LoggingServerCore.java
    - ThreadSafeMultiThreadedServer.java
Let’s Look at the Code

• Project
  – SafeMultiThreadedLoggingServer
Ungraded Assignment

• Download the code
• Read and understand how it works
• Run this code and observe its performance
  – Does this code fix the problems you saw in the MultiThreadedServer’s behavior?
  – How does this code perform relative to the MultiThreadedServer?
  – What might account for any performance differences?