CMSC 433 – Programming Language Technologies and Paradigms

Synchronization
Aspects of Synchronization

• Atomicity
  – Locking to obtain mutual exclusion
  – What we most often think about

• Visibility
  – Ensuring that changes to object fields made in one thread are seen in other threads

• Ordering
  – Ensuring that you aren’t surprised by the order in which statements are executed
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready) Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
When Are Actions Visible?

Must be the *same* lock
Forcing Visibility of Actions

- All writes from thread that holds lock M are visible to next thread that acquires lock M
  - Must be the same lock
- When accesses are unsynchronized you get no guarantees
- One effect of synchronization is to enforce visibility
"Happens before" is a partial order describing program events, invented by Leslie Lamport.
• Consider multithreaded executions as traces $R$ of events $E$

Events $E ::= \text{start}(T)$

| $\text{end}(T)$ |
| $\text{read}(T,x,v)$ |
| $\text{write}(T,x,v)$ |
| $\text{spawn}(T1,T2)$ |
| $\text{join}(T1,T2)$ |
| $\text{lock}(T,x)$ |
| $\text{unlock}(T,x)$ |

• $T$ is a thread identifier, $x$ is a variable, and $v$ is a value.
• Event $\text{read}(T,x,v)$ indicates that thread $T$ read value $v$ from variable $x$.
• Assume traces $R$ are well-formed
Happens-Before

- Let $E_1 < E_2$ be the ordering of events as they appear in the trace
- Define happens-before ordering $\ll$ in a trace $R$ as follows:

$$E_1 \ll E_2 \text{ iff } E_1 < E_2 \text{ and one of the following holds:}$$

a) $\text{thread}(E_1) = \text{thread}(E_2)$
b) $E_1$ is $\text{spawn}(T_1,T_2)$, and $E_2$ is $\text{start}(T_2)$
c) $E_2$ is $\text{join}(T_1,T_2)$, and $E_1$ is $\text{end}(T_2)$
d) $E_1$ is $\text{unlock}(T_1,x)$ and $E_2$ is $\text{lock}(T_2,x)$
e) there exists $E_3$ with $E_1 \ll E_3$ and $E_3 \ll E_2$ (i.e., the happens-before ordering is transitive)
Visibility

• For a trace $r$ containing
  – $EW == \text{write}(T1,x,v1)$ and $ER == \text{read}(T2,x,v2)$
• $EW$ "is not visible" to $ER$ if
  – $ER <: EW$
  – There exists some event $EW2 == \text{write}(T,x,v3)$ such that $EW <: EW2 <: R$ (i.e., the first write is overwritten by the second)
• Otherwise $EW$ is visible at $ER$
Example

Initially $x == 0$

Thread 1:

\[
\begin{align*}
  x &= 1 \\
  y &= 2;
\end{align*}
\]

Thread 2:

\[
\begin{align*}
  y &= x;
\end{align*}
\]

- $R1 == \text{write}(T1,x,1); \text{read}(T2,x,0); \text{write}(T2,y,0); \text{write}(T1,y,2)$
- $\text{read}(T2,x,0)$ does not happen-before $\text{write}(T1,x,1)$
- Both $x==1$ and $x==0$ are visible
Example

Initially $x == 0$

Thread 1: 
- $x = 1$
- $y = 2$

Thread 2: 
- $y = x$

- $R2 == \text{write}(T1,x,1); \underline{\text{read}}(T2,x,1); \text{write}(T2,y,1); \text{write}(T1,y,2)$
- read(T2,x,1) does not happen-before write(T1,x,1)
- Both $x == 1$ and $x == 0$ are visible
Example

Initially $x == 0$

Thread 1:  
- $x = \_\_\_$
- $y = 2$

Thread 2:  
- $y = x$

• $R3 == \textbf{read}(T2,x,0); \textbf{write}(T1,x,1); \textbf{write}(T2,y,0); \textbf{write}(T1,y,2)$
• $\textbf{read}(T2,x,0)$ goes first, so only $x==0$ is visible
Initially $x == 0$

Thread 1:  
$x = 1$
$y = 2$;

Thread 2:  
y = x;

- $R4 == \text{write}(T1,x,1); \text{read}(T2,x,1); \text{write}(T1,y,2); \text{write}(T2,y,1)$
- $\text{write}(T2,y,1)$ goes last
Trace R5

Initially \( x == 0 \)

Thread 1: 
\[
\begin{align*}
x &= 1 \\
y &= 2;
\end{align*}
\]

Thread 2: 
\[
\begin{align*}
y &= x;
\end{align*}
\]

- \( R5 == \text{read}(T2,x,0); \text{write}(T1,x,1); \text{write}(T1,y,2); \text{write}(T2,y,0) \)
- \( \text{write}(T2,y,0) \) goes last
Example

- So $y$ can end up being $\{0,1,2\}$

R1 == write(T1,x,1); read(T2,x,0); write(T2,y,0); write(T1,y,2)
R2 == write(T1,x,1); read(T2,x,1); write(T2,y,1); write(T1,y,2)
R3 == read(T2,x,0); write(T1,x,1); write(T2,y,0); write(T1,y,2)
R4 == write(T1,x,1); read(T2,x,1); write(T1,y,2); write(T2,y,1)
R5 == read(T2,x,0); write(T1,x,1); write(T1,y,2); write(T2,y,0)
Another Example

(starting with $x = 0$)

Thread 1:

lock(y);

$x = 1$;

unlock(y);

Thread 2:

lock(y);

$x = x + 1$;

unlock(y);
Another Example

(starting with $x = 0$)

Thread 1:
- lock(y);
- $x = 1$;
- unlock(y);

Thread 2:
- lock(y);
- $x = x + 1$;
- unlock(y);

R1: lock(T1,y); write(T1,x,1); unlock(T1,y); lock(T2,y); read(T2,x,1); write(T2,x,2); unlock(T2,y)
Another Example

(Starting with $x = 0$)

Thread 1:
- lock(y);
- $x = 1$;
- unlock(y);

Thread 2:
- lock(y);
- $x = x + 1$;
- unlock(y);

R2: lock(T2,y); read(T2,x,0); write(T2,x,1); unlock(T2,y);
    lock(T1,y); write(T1,x,1); unlock(T1,y);
Data Races

• The happens-before relation allows us to formally define data races

• A data race takes place when there are two events in trace R that
  – access the same memory location
  – at least one is a write
  – they are unordered according to happens-before
Data Race

Initially $x == 0$

Thread 1:
\[ \begin{align*}
x &= 1 \\
y &= 2;
\end{align*} \]

Thread 2:
\[ \begin{align*}
y &= x;
\end{align*} \]

- $R1 == \text{write}(T1,x,1); \text{read}(T2,x,0); \text{write}(T2,y,0); \text{write}(T1,y,2)$
- Happens-before
  - $\text{write}(T1,x,1) <: \text{write}(T1,y,2)$ and $\text{read}(T2,x,0) <: \text{write}(T2,y,0)$
- Data races between
  - $\text{write}(T1,x,1)$ and $\text{read}(T2,x,0)$
  - $\text{write}(T1,y,1)$ and $\text{write}(T2,y,0)$
Volatile Fields

• When a field is declared volatile the JVM ensures that all threads see the latest value for the variable
• This allows you to access a shared field without using synchronization
Using Volatile

• A one-writer/many-reader value
  – Simple control flags:
    • volatile boolean done = false;

• Keeping track of a “recent value” of something
Limitations

• Incrementing a volatile field is not atomic
  – In general, writes to a volatile field that depend on the previous value of that field don’t work

• A volatile reference to an object isn’t the same as having the fields of that object be volatile
  – No way to make elements of an array volatile

• Can’t keep two volatile fields in sync
class Test {
    static int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() { System.out.println("i=
= " + i + " j=" + j); }
}

• Thread A calls Test.one() repeatedly
• Thread B calls Test.two() repeatedly
• Can the printed value of j ever be greater than that of i?
  – Yes. This is completely unsynchronized.
Example

class Test {
    static int i = 0, j = 0;
    static synchronized void one() { i++; j++; }
    static synchronized void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}

• How about now?
  – No. i and j are updated and read in apparent textual order
Example

class Test {
    static volatile int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() { System.out.println("i=" + i + " j=" + j); }
}
• How about now?
    – j always >= i, but could be a lot bigger
    – e.g., one() could be called many times between the time two() access the value of i and then accesses the value of j.
Quiz Time

• Can this result in i=0 and j=0?
Doesn’t Seem Possible...

- But this can happen!
How Can This Happen?

• Compiler can reorder statements
  – Or keep values in registers
• Processor can reorder them
• On multi-processor systems, values not synchronized in global memory
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready) Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
Data Race

Initially $x == 0$

Thread 1: Thread 2:
$x = 1$ $y = x$
$y = 2$

- $R1 == \text{write}(T1,x,1); \text{read}(T2,x,0); \text{write}(T2,y,0); \text{write}(T1,y,2)$
- Happens-before
  - $\text{write}(T1,x,1) <: \text{write}(T1,y,2)$
  - $\text{read}(T2,x,0) <: \text{write}(T2,y,0)$
- Compiler might reorder the writes to $x$ and $y$ because they are independent and not ordered with accesses outside $T1$
Ordering

- Synchronization also influences the ordering of statements
Synchronization not a Panacea

- Two threads can block on locks held by the other; this is called *deadlock*
  - A set of threads is *deadlocked* if each thread is waiting for an event that only another thread in the set (including itself) can cause.

```java
Object A = new Object();
Object B = new Object();
T1.run() {
    synchronized (A) {
        synchronized (B) {
            ...
        }
    }
}
T2.run() {
    synchronized (B) {
        synchronized (A) {
            ...
        }
    }
}
```
Deadlock

• Quite possible to create code that deadlocks
  – Thread 1 holds lock on A
  – Thread 2 holds lock on B
  – Thread 1 is blocked trying to acquire lock on B
  – Thread 2 is blocked trying to acquire lock on A
  – Deadlock!

• Not easy to detect when deadlock has occurred
  – Other than by the fact that nothing is happening
Deadlock Example

Object A = new Object();
Object B = new Object();
T1.run() {
    synchronized (A) {
        synchronized (B) {
            ...
        }
    }
}
T2.run() {
    synchronized (B) {
        synchronized (A) {
            ...
        }
    }
}
• For deadlock to occur the following conditions must hold simultaneously

1. Mutual exclusion: a non-sharable resource exists
2. Hold and wait: processes already holding resources may request new resources held by other processes
3. No preemption: No resource can be forcibly removed from a process holding it
4. Circular wait: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds
Deadlock: Wait graphs

Thread T1 holds lock A

Thread T2 attempting to acquire lock B

Deadlock occurs when there is a cycle in the graph
Wait graph example

T1 holds lock on A
T2 holds lock on B
T1 is trying to acquire a lock on B
T2 is trying to acquire a lock on A
Key Ideas

- **Multiple threads can run simultaneously**
  - Either truly in parallel on a multiprocessor
  - Or effectively in parallel on a single processor
    - Assuming a running thread can be preempted at any time

- **Threads can share data**
  - Need to prevent interference
    - Synchronization, immutability, and other methods
  - Overuse use of synchronization can create deadlock
    - Violation of liveness
Ungraded Assignment

• Repo
  – https://bitbucket.org/cmsc433_spring2012/codeexamples/src

• Project
  – DeadLockExample

• This code can deadlock. See if you can figure out why.
Ungraded Assignment

- x=y=0
- x=1
- y=x
- x=y=0
- y=1
- j=y

- Work out the happens-before relations for this code
- Use the happens-before relation to show the existence of data races