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;
    }
}

• Some possible answers
  a) 42
  b) 0
  c) Nothing
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 $E ::= start(T) \ (i.e., \ T.start())$

- $end(T)$
- $read(T,x,v)$
- $write(T,x,v)$
- $spawn(T1,T2) \ (i.e., \ in \ T1: \ T2 = \text{new} \ \text{Thread();})$
- $join(T1,T2) \ (i.e., \ in \ T1: \ T2.join();)$
- $lock(T,x)$
- $unlock(T,x)$

Where $T$ is a thread identifier, $x$ is a variable, and $v$ is a value.

- Event $read(T,x,v)$ indicates that thread $T$ read value $v$ from variable $x$.
- Assume traces $R$ are well-formed
Let $E_1 < E_2$ be the ordering of events as they appear in the trace
Define happens-before ordering $\langle: \rangle$ in a trace $R$ as follows:

\[ E_1 \langle: E_2 \text{ iff } E_1 < E_2 \text{ and one of the following holds:} \]

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 $E_1 \langle: E_3$ and $E_3 \langle: E_2$ (i.e., the happens-before ordering is transitive)
Visibility

• For a trace \( R \) containing
  – \( WE== \text{write}(T1,x,v1) \) and \( RE== \text{read}(T2,x,v2) \)
• \( WE \) "is not visible" to \( RE \) if
  – \( RE <: WE \)
  – There exists some event \( WE2== \text{write}(T,x,v3) \) such that \( WE<: WE2<: RE \) (i.e., the first write is overwritten by the second)
• Otherwise \( WE \) is visible at \( ER \)
Trace R1

Initially $x == 0$

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

Thread 2: 
$y = x$

- $R1 == write(T1,x,1); read(T2,x,0); write(T2,y,0); write(T1,y,2)$
- $read(T2,x,0)$ does not happen-before $write(T1,x,1)$
- So other behaviors are possible
  - For example, both $x==1$ and $x==0$ are visible
Trace R2

Initially x == 0

Thread 1:  
x = 1  
y = 2;

Thread 2:  
y = x;

- R2 == write(T1,x,1); read(T2,x,1); write(T2,y,1); write(T1,y,2)
- read(T2,x,1) does not happen-before write(T1,x,1)
- Both x==1 and x==0 are visible
Trace R3

Initially $x == 0$

Thread 1: Thread 2:
\[
x = 1 \quad y = x;
\]
\[
y = 2; \quad y = 2;
\]

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

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

Thread 2:
- $y = x$

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

Initially $x == 0$

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

Thread 2:
$y = x$;

- $R5 == read(T2,x,0); write(T1,x,1); write(T1,y,2); write(T2,y,0)$
- $write(T2,y,0)$ goes last
Example

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

\[
\begin{align*}
R_1 &= \text{write}(T_1, x, 1); \text{read}(T_2, x, 0); \text{write}(T_2, y, 0); \text{write}(T_1, y, 2) \\
R_2 &= \text{write}(T_1, x, 1); \text{read}(T_2, x, 1); \text{write}(T_2, y, 1); \text{write}(T_1, y, 2) \\
R_3 &= \text{read}(T_2, x, 0); \text{write}(T_1, x, 1); \text{write}(T_2, y, 0); \text{write}(T_1, y, 2) \\
R_4 &= \text{write}(T_1, x, 1); \text{read}(T_2, x, 1); \text{write}(T_1, y, 2); \text{write}(T_2, y, 1) \\
R_5 &= \text{read}(T_2, x, 0); \text{write}(T_1, x, 1); \text{write}(T_1, y, 2); \text{write}(T_2, y, 0)
\end{align*}
\]
Another Example

(\textit{starting with } x == 0)

Thread 1:
\begin{itemize}
  \item lock(y);
  \item x = 1;
  \item unlock(y);
\end{itemize}

Thread 2:
\begin{itemize}
  \item lock(y);
  \item x = x + 1;
  \item unlock(y);
\end{itemize}
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:
  $x = 1$
  $y = 2$

Thread 2:
  $y = x$

- R1 == write(T1,x,1); read(T2,x,0); write(T2,y,0); write(T1,y,2)
- Happens-before
  - write(T1,x,1) <: write(T1,y,2) and read(T2,x,0) <: write(T2,y,0)
- Data races between
  - write(T1,x,1) and read(T2,x,0)
  - write(T1,y,1) and 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
Example

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
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 could be < i, or it could be a lot bigger
  – I is incremented in one(). Both i and j are printed in two() before j
    is incremented in one()
  – e.g., one() could be called many times between the time two() accesses 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
Data Race

Initially $x == 0$

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

Thread 2:
- $y = x$

- R1 == write(T1,x,1); read(T2,x,0); write(T2,y,0); write(T1,y,2)
- Happens-before
  - write(T1,x,1) <: write(T1,y,2)
  - read(T2,x,0) <: 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
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.
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
Object A = new Object();
Object B = new Object();

T1.run() {
    synchronized (A) {
        synchronized (B) {
            ...
        }
    }
}

T2.run() {
    synchronized (B) {
        synchronized (A) {
            ...
        }
    }
}
Deadlock Conditions

• 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 that holds 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

- Project
  - DeadLockExample

- This code can deadlock. See if you can figure out why.
In-Class Assignment

Initially \( x=y=0 \)

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

• Write out 4 traces that could occur given this program
• Work out the happens-before relations for this code
• Use the happens-before relation to prove the existence of one or more data races