Concurrency Issues

- Threads allow concurrent activities, which can be both good and bad!
- Two opposing design forces
  - Safety: “Nothing bad ever happens.”
  - Liveness: “Something (useful) eventually happens.”
- A safe system may not be live and a live system may not be safe. Balance is key.

Violating Safety

- Data can be shared by threads
  - Scheduler can interleave or overlap threads arbitrarily
  - Can lead to interference
    - Storage corruption (e.g. a data race/race condition)
    - Violation of representation invariant
    - Violation of a protocol (e.g. A occurs before B)

Systems = Objects + Activities

- Safety is a property of objects, and groups of objects, that participate across multiple activities.
  - Can be a concern at many different levels: objects, composites, components, subsystems, hosts, ...
- Liveness is a property of activities, and groups of activities, that span across multiple objects.
  - Levels: Messages, call chains, threads, sessions, scenarios, script workflows, use cases, transactions, data flows, mobile computations, ...

Safe Objects

- 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

Data Race Example

```java
public class Example extends Thread {
    private static int cnt = 0;  // shared state
    public void run() {
        int y = cnt;
        cnt = y + 1;
    }
    public static void main(String[] args) {
        Thread t1 = new Example();
        Thread t2 = new Example();
        t1.start();
        t2.start();
    }
}
```

Data Race Example

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

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

```java
static int cnt = 0;  // Shared state cnt = 0
t1.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}
t2.run() {
    int y = cnt;  // T1 executes, grabbing
    cnt = y + 1;  the global counter value into y.
}
```

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

Data Race Example

```java
static int cnt = 0;  // Shared state cnt = 1
t1.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}
t2.run() {
    int y = cnt;  // T2 completes. T1
    cnt = y + 1;  executes again, storing the
                the old counter value (1) rather  
                than the new one (2)!
}
```

But When I Run it Again?

```java
static int cnt = 0;  // Shared state cnt = 0
t1.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}
t2.run() {
    int y = cnt;  // T1 is pre-empted. T2
    cnt = y + 1;  executes, grabbing the global
                counter value into y.
}
```

T1 executes, grabbing
the global counter value into y.
Data Race Example

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

T1 executes, grabbing the global counter value into y.

Data Race Example

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

T1 executes again, storing the counter value

What happened?

• In the first example, T1 was preempted after it read the counter but before it stored the new value.
  – Depends on the idea of an atomic action
  – Violated an object invariant

• A particular way in which the execution of two threads is interleaved is called a schedule. We want to prevent this undesirable schedule.

• Undesirable schedules can be hard to reproduce, and so hard to debug.

Question

• 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

• Race conditions are hard to find
Avoiding Interference: Synchronization

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

Lock, for protecting The shared state
Acquires the lock; Only succeeds if not held by another thread
Releases the lock

Applying synchronization

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

Shared state cnt = 0

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

Applying synchronization

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

T1 reads cnt into y

Applying synchronization

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

T1 runs, assigning to cnt

Applying synchronization

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

T1 releases the lock and terminates
Applying synchronization

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

Shared state  cnt = 1

Applying synchronization

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

Shared state  cnt = 1

Applying synchronization

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

Shared state  cnt = 2

Locks

- Any Object subclass has (can act as) a lock
- Only one thread can hold the lock on an object
  - other threads block until they can acquire it
- If a thread already holds the lock on an object
  - The thread can reacquire the same lock many times
  - Lock is released when object unlocked the corresponding number of times
- No way to only attempt to acquire a lock
  - Either succeeds, or blocks the thread

Synchronized statement

- `synchronized (obj) { statements }`
- Obtains the lock on `obj` before executing statements in block
- Releases the lock when the statement block completes
  - Either normally, or do to a return, break, or exception being thrown in the block

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. `Fonclass`
  - Not the same as `this`!
### Synchronization Style

- **Design decision**
  - **Internal synchronization (class is thread-safe)**
    - Have a stateful object synchronize itself (e.g. with synchronized methods)
  - **External synchronization (class is thread-compatible)**
    - Have callers perform synchronization before calling the object
- Can go both ways:
  - **Thread-safe: Random**
  - **Thread-compatible: ArrayList, HashMap, …**

### Synchronization not a Panacea

- Two threads can block on locks held by the other; this is called **deadlock**

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

### 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

### Deadlock

- Quite possible to create code that deadlocks
  - Thread 1 holds lock on A
  - Thread 2 holds lock on B
  - Thread 1 is trying to acquire a lock on B
  - Thread 2 is trying to acquire a lock on A
  - Deadlock!
- Not easy to detect when deadlock has occurred
  - other than by the fact that nothing is happening

### Key Ideas

- Multiple threads can run simultaneously
  - Either truly in parallel on a multiprocessor
  - Or can be scheduled on a single processor
    - A running thread can be pre-empted at any time
- Threads can share data
  - In Java, only fields can be shared
    - Need to prevent interference
      - Synchronization is one way, but not the only way
      - Overuse of synchronization can create deadlock
    - Violation of liveness
Guaranteeing Safety

- Ensure objects are accessible only when in a consistent and appropriate state
  - All invariants are maintained
  - Presents subclass obligations

Guaranteeing Liveness

- Ensuring availability of services
  - Called methods eventually execute
- Ensuring progress of activities
  - Managing resource contention
  - Freedom from deadlock
  - Fairness
  - Fault tolerance