Pick From These 3 Choices for Memory:

For every **memory location** in your program (e.g., object field), you must obey at least one of the following:

1. **Thread-local**: Do not use the location in > 1 thread
2. **Immutable**: Do not write to the memory location
3. **Synchronized**: Use synchronization to control access
Thread-Local

Whenever possible, do not share resources

- Easier for each thread have its own thread-local copy of a resource instead of one with shared updates

- Correct only if threads do not communicate through resource
  - In other words, multiple copies are a correct approach
  - Example: Random objects

- Note:
  Because each call-stack is thread-local, never need to synchronize on local variables

*In typical concurrent programs, the vast majority of objects should be thread-local: shared-memory usage should be minimized*
**Immutable**

Whenever possible, do not update objects
- Make new objects instead

One of the key tenets of *functional programming* (see CSE 341)
- Generally helpful to avoid *side-effects*
- Much more helpful in a concurrent setting

If a location is only read, never written, no synchronization needed
- Simultaneous reads are *not* races and *not* a problem

*In practice, programmers usually over-use mutation* – *minimize it*
Everything Else: Keep it Synchronized

After minimizing the amount of memory that is both (1) thread-shared and (2) mutable, we need guidelines for how to use locks to keep that data consistent.

Guideline #0: No data races
• Never allow two threads to read/write or write/write the same location at the same time

Necessary:
  In Java or C, a program with a data race is almost always wrong

But Not Sufficient:
  Our peek example had no data races
**Consistent Locking**

**Guideline #1: Consistent Locking**

For each location that requires synchronization, have a lock that is always held when reading or writing the location

- We say the lock **guards** the location

- The same lock can guard multiple locations (and often should)

- Clearly document the guard for each location

- In Java, the guard is often the object containing the location
  - **this** inside object methods
  - But also common to guard a larger structure with one lock to ensure mutual exclusion on the structure
Consistent Locking

- The mapping from locations to guarding locks is conceptual, and must be enforced by you as the programmer.
- It partitions the shared-&-mutable locations into “which lock”

Consistent locking is:

*Not Sufficient:*  
It prevents all data races, but still allows bad interleavings
  - Our *peek* example used consistent locking, but had exposed intermediate states and bad interleavings

*Not Necessary:*  
Can dynamically change the locking protocol
Beyond Consistent Locking

• Consistent locking is an excellent guideline
  – A “default assumption” about program design
  – You will save yourself many a headache using this guideline

• But it is not required for correctness:
  Different program phases can use different locking techniques
  – Provided all threads coordinate moving to the next phase

• Example from Project 3 Version 5:
  – A shared grid being updated, so use a lock for each entry
  – But after the grid is filled out, all threads except 1 terminate
    • So synchronization no longer necessary (i.e., thread local)
  – And later the grid is only read in response to queries
    • Makes synchronization doubly unnecessary (i.e., immutable)
Lock Granularity

Coarse-Grained: Fewer locks (i.e., more objects per lock)
- Example: One lock for entire data structure (e.g., array)
- Example: One lock for all bank accounts

Fine-Grained: More locks (i.e., fewer objects per lock)
- Example: One lock per data element (e.g., array index)
- Example: One lock per bank account

“Coarse-grained vs. fine-grained” is really a continuum
Trade-Offs

Coarse-grained advantages
- Simpler to implement
- Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
- Much easier to implement modifications of data-structure shape

Fine-grained advantages
- More simultaneous access (improves performance when coarse-grained would lead to unnecessary blocking)

Guideline #2: Lock Granularity
Start with coarse-grained (simpler), move to fine-grained (performance) only if contention on coarse locks is an issue. Alas, often leads to bugs.
Example: Separate Chaining Hashtable

- Coarse-grained: One lock for entire hashtable
- Fine-grained: One lock for each bucket

Which supports more concurrency for **insert** and **lookup**?

  Fine-grained; allows simultaneous access to diff. buckets

Which makes implementing **resize** easier?

  Coarse-grained; just grab one lock and proceed
  - How would you do it?

Maintaining a **numElements** field will destroy the potential benefits of using separate locks for each bucket, why?

  Updating it each insert w/o a coarse lock would be a data race
Critical-Section Granularity

A second, orthogonal granularity issue is critical-section size
  – How much work to do while holding lock(s)

If critical sections run for too long:
  – Performance loss because other threads are blocked

If critical sections are too short:
  – Bugs because you broke up something where other threads should not be able to see intermediate state

Guideline #3: Granularity
  Do not do expensive computations or I/O in critical sections, but also do not introduce race conditions
Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table
  – Assume lock guards the whole table

Papa Bear’s critical section was too long (table locked during expensive call)

```java
synchronized(lock) {
    v1 = table.lookup(k);
    v2 = expensive(v1);
    table.remove(k);
    table.insert(k,v2);
}
```
Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume `lock` guards the whole table

```java
synchronized(lock) {
    v1 = table.lookup(k);
}

v2 = expensive(v1);

synchronized(lock) {
    table.remove(k);
    table.insert(k,v2);
}
```

Mama Bear’s critical section was too short

(if another thread updated the entry, we will lose an update)
Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume `lock` guards the whole table

```
done = false;
while(!done) {
    synchronized(lock) {
        v1 = table.lookup(k);
    }
    v2 = expensive(v1);
    synchronized(lock) {
        if(table.lookup(k)==v1) {
            done = true;
            table.remove(k);
            table.insert(k,v2);
        }
    }
}
```

Baby Bear’s critical section was just right

(if another update occurred, try our update again)
Atomicity

An operation is atomic if no other thread can see it partly executed
- Atomic as in “appears indivisible”
- Typically want ADT operations atomic, even to other threads running operations on the same ADT

Guideline #4: Atomicity
- Think in terms of what operations need to be atomic
- Make critical sections just long enough to preserve atomicity
- Then design locking protocol to implement the critical sections

In other words:
Think about atomicity first and locks second
Do Not Roll Your Own

• It is rare that you should write your own data structure
  – Excellent implementations provided in standard libraries
  – Point of CSE 332 is to understand the key trade-offs, abstractions, and analysis of such implementations

• Especially true for concurrent data structures
  – Far too difficult to provide fine-grained synchronization without race conditions
  – Standard thread-safe libraries like ConcurrentHashMap written by world experts

Guideline #5: Libraries
Use built-in libraries whenever they meet your needs
Motivating Memory-Model Issues

Tricky and *surprisingly wrong* unsynchronized concurrent code

```java
class C {
   private int x = 0;
   private int y = 0;

   void f() {
      x = 1;
      y = 1;
   }

   void g() {
      int a = y;
      int b = x;
      assert(b >= a);
   }
}
```

First understand why it looks like the assertion cannot fail:

- Easy case: call to `g` ends before any call to `f` starts
- Easy case: at least one call to `f` completes before call to `g` starts
- If calls to `f` and `g` *interleave*…
Interleavings are Not Enough

There is no interleaving of $f$ and $g$ where the assertion fails

- Proof #1: Exhaustively consider all possible orderings of access to shared memory (there are 6)
- Proof #2:
  If $! (b \geq a)$, then $a==1$ and $b==0$.
  But if $a==1$, then $y=1$ happened before $a=y$.
  Because programs execute in order:
    $a=y$ happened before $b=x$ and $x=1$ happened before $y=1$.
  So by transitivity, $b==1$. Contradiction.

\begin{align*}
\text{Thread 1: } f & \quad \text{Thread 2: } g \\
\text{x = 1; } & \quad \text{int } a = y; \\
\text{y = 1; } & \quad \text{int } b = x; \\
\text{} & \quad \text{assert}(b \geq a); \\
\end{align*}
Wrong

However, the code has a *data race*

- Unsynchronized read/write or write/write of same location

If code has data races, you cannot reason about it with interleavings

- This is simply the rules of Java (and C, C++, C#, other languages)
- Otherwise we would slow down all programs just to “help” those with data races, and that would not be a good engineering trade-off
- So the assertion can fail
Why

For performance reasons, the compiler and the hardware will often reorder memory operations

- Take a compiler or computer architecture course to learn more

Of course, we cannot just let them reorder anything they want

- Each thread computes things by executing code in order
- Consider: \( x = 17; \ y = x; \)
The Grand Compromise

The compiler/hardware will never perform a memory reordering that affects the result of a single-threaded program.

The compiler/hardware will never perform a memory reordering that affects the result of a data-race-free multi-threaded program.

So: If no interleaving of your program has a data race, then you can *forget about all this reordering nonsense*: the result will be equivalent to some interleaving.

Your job: Avoid data races.

Compiler/hardware job: Give illusion of interleaving *if you do your job*.
Fixing Our Example

- Naturally, we can use synchronization to avoid data races
  - Then, indeed, the assertion cannot fail

```java
class C {
    private int x = 0;
    private int y = 0;
    void f() {
        synchronized(this) { x = 1; }
        synchronized(this) { y = 1; }
    }
    void g() {
        int a, b;
        synchronized(this) { a = y; }
        synchronized(this) { b = x; }
        assert(b >= a);
    }
}
```
A Second Fix: Stay Away from This

- Java has **volatile** fields: accesses do not count as data races
  - But you cannot read-update-write

```java
class C {
    private volatile int x = 0;
    private volatile int y = 0;
    void f() {
        x = 1;
        y = 1;
    }
    void g() {
        int a = y;
        int b = x;
        assert(b >= a);
    }
}
```

- Implementation: slower than regular fields, faster than locks
- Really for experts: avoid them; use standard libraries instead
- And why do you need code like this anyway?
Code That is Wrong

• Here is a more realistic example of code that is wrong
  – No guarantee Thread 2 will ever stop (as there is a data race)
  – But honestly it will “likely work in practice”

```java
class C {
    boolean stop = false;
    void f() {
        while(!stop) {
            // draw a monster
        }
    }
    void g() {
        stop = didUserQuit();
    }
}
```

Thread 1: f()
Thread 2: g()
Motivating Deadlock Issues

Consider a method to transfer money between bank accounts:

```java
class BankAccount {
    ...
    synchronized void withdraw(int amt) {...}
    synchronized void deposit(int amt) {...}
    synchronized void transferTo(int amt, BankAccount a) {
        this.withdraw(amt);
        a.deposit(amt);
    }
}
```

Notice during call to `a.deposit`, thread holds **two** locks — Need to investigate when this may be a problem
The Deadlock

Suppose \( x \) and \( y \) are fields holding accounts

Thread 1: \( x\).transferTo(1, y)

- acquire lock for \( x \)
- do withdraw from \( x \)
- block on lock for \( y \)

Thread 2: \( y\).transferTo(1, x)

- acquire lock for \( y \)
- do withdraw from \( y \)
- block on lock for \( x \)
The Dining Philosophers

• 5 philosophers go out to dinner together at an Italian restaurant
• Sit at a round table; one fork per setting
• When the spaghetti comes, each philosopher proceeds to grab their right fork, then their left fork, then eats
• ‘Locking’ for each fork results in a **deadlock**
Deadlock

A deadlock occurs when there are threads $T_1, \ldots, T_n$ such that:

- For $i = 1, \ldots, n-1$, $T_i$ is waiting for a resource held by $T_{(i+1)}$
- $T_n$ is waiting for a resource held by $T_1$

In other words, there is a cycle of waiting
  - Can formalize as a graph of dependencies with cycles bad

Deadlock avoidance in programming amounts to techniques to ensure a cycle can never arise
Back to Our Example

Options for deadlock-proof transfer:

1. Make a smaller critical section: `transferTo` not synchronized
   - Exposes intermediate state after `withdraw` before `deposit`
   - May be okay, but exposes wrong total amount in bank

2. Coarsen lock granularity:
   one lock for all accounts allowing transfers between them
   - Works, but sacrifices concurrent deposits/withdrawals

3. Give every bank-account a unique number
   and always acquire locks in the same order
   - *Entire program* should obey this order to avoid cycles
   - Code acquiring only one lock can ignore the order
class BankAccount {
    ...
    private int acctNumber; // must be unique
    void transferTo(int amt, BankAccount a) {
        if(this.acctNumber < a.acctNumber) {
            synchronized(this) {
                synchronized(a) {
                    this.withdraw(amt);
                    a.deposit(amt);
                }
            }
        } else {
            synchronized(a) {
                synchronized(this) {
                    this.withdraw(amt);
                    a.deposit(amt);
                }
            }
        }
    }
}
StringBuffer Example

From the Java standard library

class StringBuffer {
    private int count;
    private char[] value;
...
    synchronized append(StringBuffer sb) {
        int len = sb.length();
        if(this.count + len > this.value.length)
            this.expand(...);
        sb.getChars(0,len,this.value,this.count);
    }
    synchronized getChars(int x, int y, char[] a, int z) {
        "copy this.value[x..y] into a starting at z"
    }
}
Two Problems

Problem #1:
Lock for sb not held between calls to sb.length and sb.getChars
– So sb could get longer
– Would cause append to throw an ArrayBoundsException

Problem #2:
Deadlock potential if two threads try to append in opposite directions, identical to the bank-account first example

Not easy to fix both problems without extra copying:
– Do not want unique ids on every StringBuffer
– Do not want one lock for all StringBuffer objects

Actual Java library: fixed neither (left code as is; changed documentation)
– Up to clients to avoid such situations with own protocols
**Perspective**

- Code like account-transfer and string-buffer append are difficult to deal with for deadlock

- Easier case: different types of objects
  - Can document a fixed order among types
  - Example: “When moving an item from the hashtable to the work queue, never try to acquire the queue lock while holding the hashtable lock”

- Easier case: objects are in an acyclic structure
  - Can use the data structure to determine a fixed order
  - Example: “If holding a tree node’s lock, do not acquire other tree nodes’ locks unless they are children in the tree”