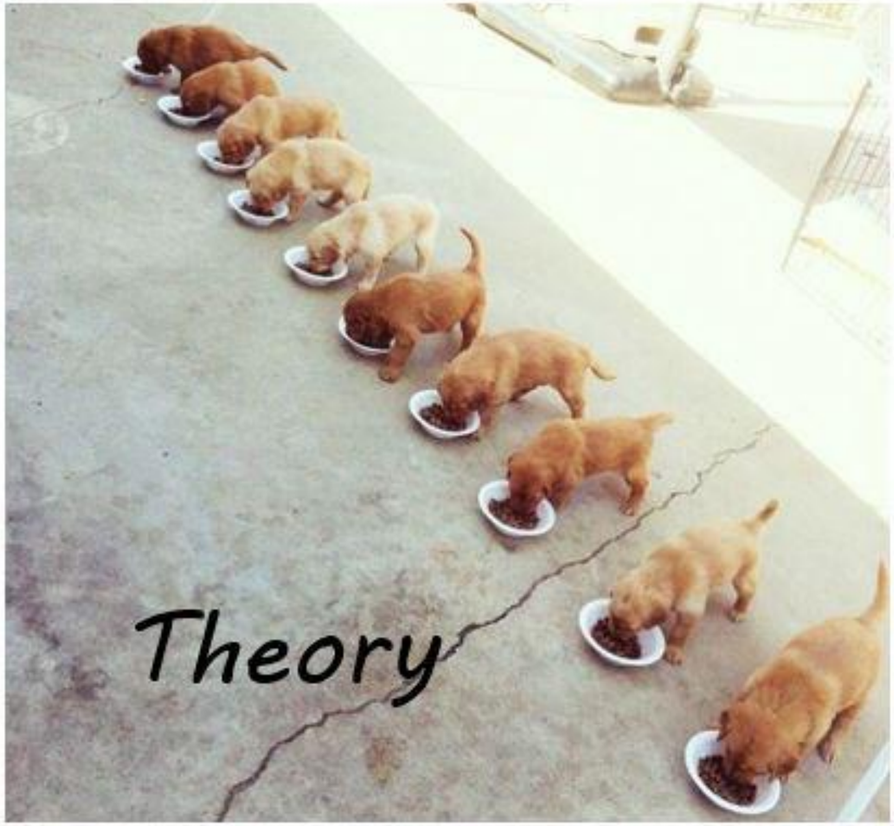
Parallel and concurrent programming 7.

Part II: Concurrency and Mutual Exclusion

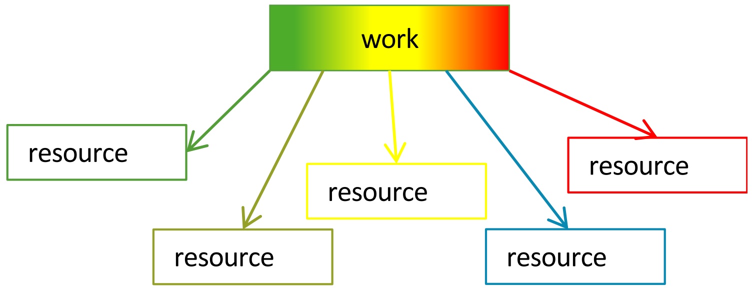
Michelle Kuttel

parallel
concurrent
Amdahl's Law
thread of control
processors versus processes
protection
data race
synchronization
THREAD
SAFETY
correctness
MUTUAL EXCLUSION
locks
readers-writers problem
liveness
DEADLOCK
starvation
HIGH PERFORMANCE COMPUTING
producer-consumer problem
EXECUTORS
thread pools
timing
Dining philosphoers problem

Multithreaded programming



Parallel programming

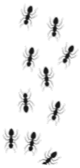


So far focused on parallel algorithms

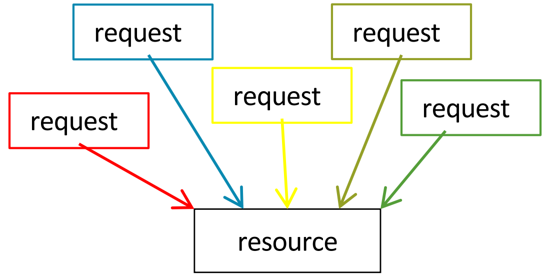
- used fork-join only
  - Lower *span* via parallel tasks
  - threads all do the same thing, really, on different data
  - **“data decomposition”**

Algorithms all had a very *simple structure* to avoid race conditions

- Each thread had memory “only it accessed”
  - Example: array sub-range
- On **`fork`**, allocated some of its memory to subthread and did not access that memory again until after **`join`** on the subthread
  - Only form of synchronization used, but vital for fork-join library.



Now, concurrent programming



Threads assigned *independent* tasks

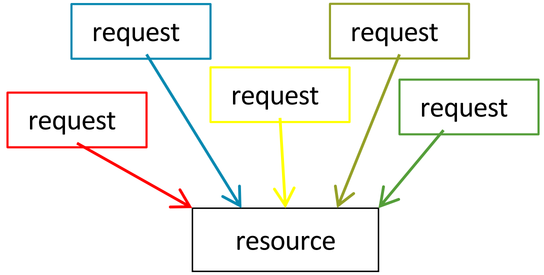
- access same resources (rather than implementing the same algorithm)
- **“task decomposition”**

Concurrency: Correctly and efficiently managing access to shared resources from multiple (possibly simultaneous) clients

More **complex structure** for algorithms

- multiple threading doing different things with data
- e.g. animator threads, threads that update data.

Now, concurrent programming



Threads assigned *independent* tasks

- access same resources (rather than implementing the same algorithm)
- **"task decomposition"**

Concurrency: Correctly and efficiently managing access to shared resources from multiple (possibly simultaneous) clients

More **complex structure** for algorithms

- multiple threading doing different things with data
- e.g. animator threads, threads that update data.

Race conditions much more prevalent!

Concurrent programming

Unlike parallelism, not all about implementing algorithms faster. :

- ***Different tasks*** *may* ***need to happen at once*** *for program to work.*
  - Example: Animation in a game
- *Simultaneous tasks may be necessary for* ***responsiveness***
  - Example: Respond to GUI events in one thread while another thread is performing an expensive computation
  - Web server
- *Simultaneous tasks may be necessary for* ***efficient processor utilization***
  - If 1 thread “goes to disk,” have something else to do (often called “latency hiding”)
- *Simultaneous tasks may be necessary for* ***failure isolation***
  - Convenient structure if want to *interleave* multiple tasks and don’t want an exception in one to stop the other

Synchronization

Synchronization constraints are **requirements** pertaining to the **order of events**.

This is the programmer's responsibility to ensure that synchronization constraints are enforced

- "The compiler" has no idea what interleavings should or shouldn't be allowed in your program
- you need language primitives to do it!

Simple example:

- **Signaling**:

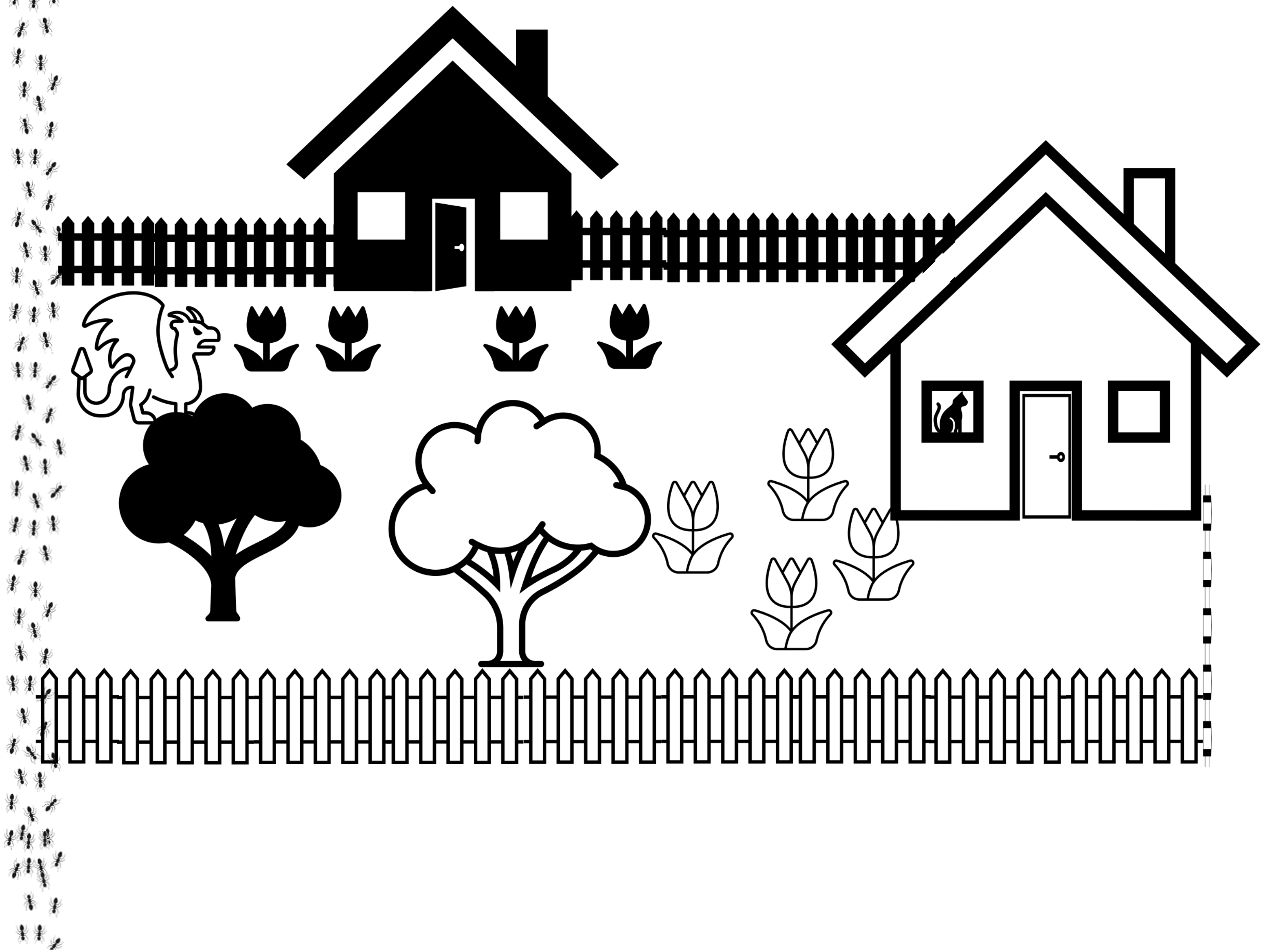
  Event A must happen ***before*** Event B.

  - We've seen this already – solved with `join`.
    - Not a universal solution – why?

Synchronization

- In **real life** we typically use a **clock** to enforce synchronization constraints.
  - How do we know if A happened before B?
  - If we know what time both events occurred, we can just compare the times.
- In computer systems, we need to satisfy synchronization constraints without a clock
  - there is **no universal clock across different computers**
  - we don't know with fine enough **resolution** when events occur.
    - (clock ticks are too far apart)

A Mutual Exclusion Fable:
Alice and Bob share a garden… and have pets



Mutual exclusion:

Event A
and Event B
must **not**
happen ***at the same time***.

The Mutual Exclusion problem

One of the most important synchronization problems in concurrent programming

- several processes compete for a resource, but the nature of the resource **requires** that only one process at a time actually accesses the resource at any point
- use **synchronization** to avoid **incorrect simultaneous access**

Also called the **critical section problem.**

Abstraction of many synchronization issues.

If two processes try to enter a **critical section**, one process must win and the other must **block:**

- not proceed until the "winning" process has completed

Examples of where mutual exclusion is necessary

Multiple threads:

1. Processing different bank-account operations
   - What if 2 threads change the same account at the same time?

2. Using a shared cache (e.g., hash table) of recent files
   - What if 2 threads insert the same file at the same time?

3. Creating a pipeline (like an assembly line) with a queue for handing work to next thread in sequence
   - What if enqueuer and dequeuer adjust a circular array queue at the same time?

Simple example: counter shared between threads

```
public class Counter {
    private long value;

    Counter() { value=0; }

    public long get() {return value;}

    public void set(long newVal) {
        value=newVal;
    }

    public void incr() {value++;}
}
```

Procedure for Thread *i*

```
public class CounterUpdateThread extends Thread {
	Counter sharedCount;
	int rep;
	CounterUpdateThread(Counter c,int repeats) {
		sharedCount= c;
		rep=repeats;
	}
	public void run() {
		for (int i=0;i<rep;i++) {
				sharedCount.incr();

		}
	}
}
```

Shared counter object

Testing it all

```
public class TestCounterSafety {
	public static void main(String args[]) throws InterruptedException
{
		int noThrds = 100;
		int addPerThread=100;
		Counter sharedCount= new Counter();

		CounterUpdateThread [] thrds= new CounterUpdateThread[noThrds];
		for (int i=0;i<noThrds;i++) {
			thrds[i]=new CounterUpdateThread(sharedCount,addPerThread);
		}
		for (int i=0;i<noThrds;i++) {
			thrds[i].start();
		}

		for (int i=0;i<noThrds;i++) {
			thrds[i].join();
		}

		int expectedVal = noThrds*addPerThread;
		System.out.println("Final value of counter is:"
			+ sharedCount.get() + " and should be:" + expectedVal);
		}
}
```

Testing it all

```
public class TestCounterSafety {
	public static void main(String args[]) throws InterruptedException
{
		int noThrds = 100;
		int addPerThread=100;
		Counter sharedCount= new Counter();

		CounterUpdateThread [] thrds= new CounterUpdateThread[noThrds];
		for (int i=0;i<noThrds;i++) {
			thrds[i]=new CounterUpdateThread(sharedCount,addPerThread);
		}
		for (int i=0;i<noThrds;i++) {
			thrds[i].start();
		}

		for (int i=0;i<noThrds;i++) {
			thrds[i].join();
		}

		int expectedVal = noThrds*addPerThread;
		System.out.println("Final value of counter is:"
			+ sharedCount.get() + " and should be:" + expectedVal);
		}
}
```

Race condition!

Simple example: counter shared between threads

```java
public class Counter {
        private long value;

        Counter() { value=0; }

        public long get() {return value;}

        public void set(long newVal) {
                value=newVal;
        }

        public void incr() {value++;}
}
```

OK for single thread,
not for concurrent threads

What It Means

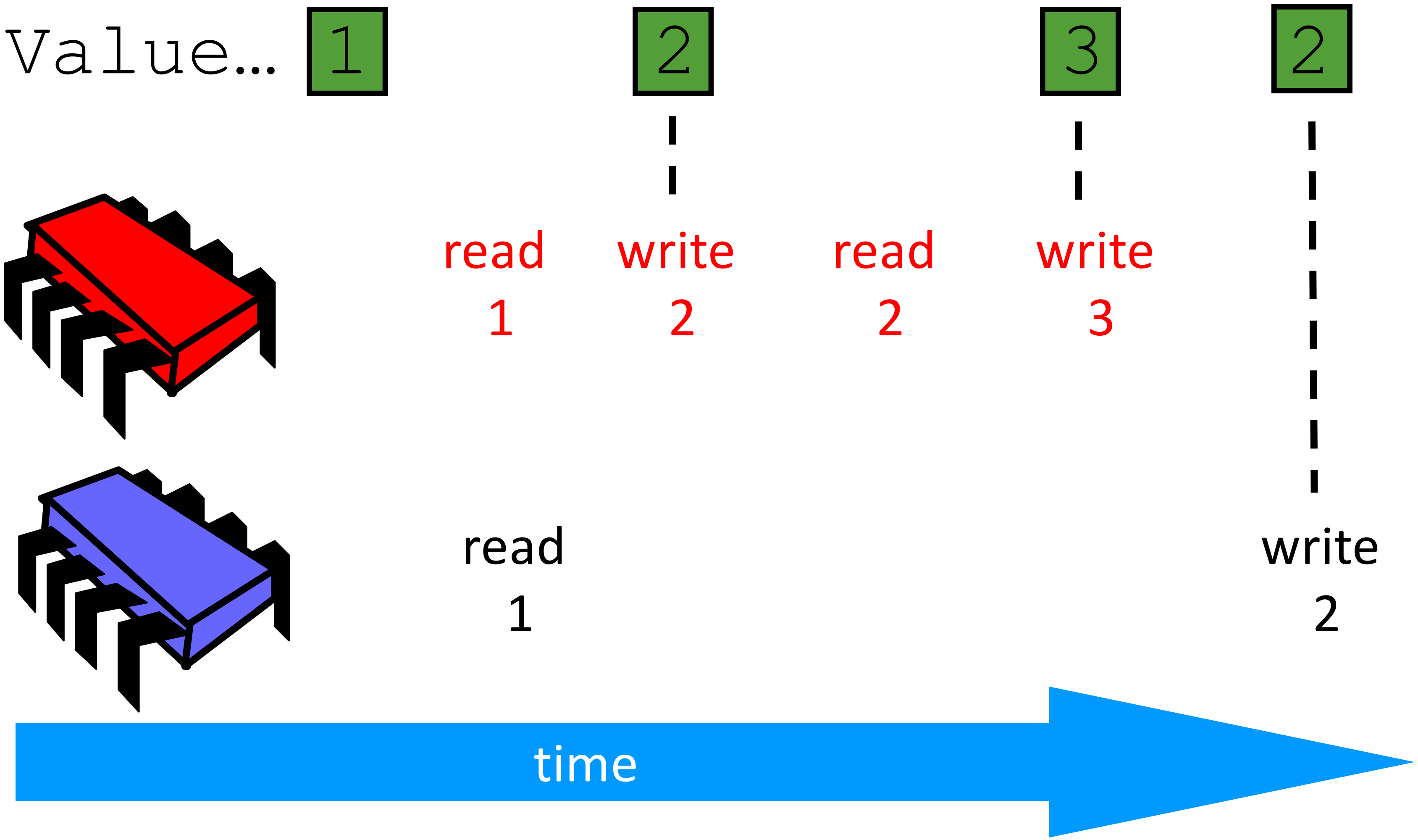
```
public class Counter {
	private long value;
//[...]
	public void incr() {value++;}
}
```

What It Means

```
public class Counter {
        private long value;
//[...]
        public void incr() {value++;}
}
```

```
temp   = value;
temp = temp + 1;
return temp;
```

Not so good…



Solving Mutual exclusion in Java

`join` is not what we want

- We need to block until another thread is “done using what we need” not “completely done executing”

We need atomic actions

Operations A and B are **atomic** with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has.

The operation is *indivisible.*

What It Means

```
public class Counter {
        private long value;
//[...]
        public void incr() {value++;}
}
```

```
temp  = value;
temp = temp + 1;
return temp;
```

Make these steps *atomic* (indivisible)

How do we do this?

Mutual exclusion: Atomic classes in Java

One way we can fix this is with a thread-safe **atomic variable class:** `java.util.concurrent` contains **atomic variable classes** for atomic actions on numbers and objects.

- AtomicInteger, AtomicLong, AtomicBoolean, AtomicReference
- **note: Integer NOT atomic by default.**
- **no atomic classes** for character and double.
- all have `getAndSet()` method that atomically sets the value and returns it

So, we can replace a long counter with `AtomicLong` to ensure that all updates to the counter state are **atomic**.

CORRECT Procedure for Thread *i*

```
public class Counter {
  private AtomicLong value;  // this is a class!

  Counter() { value=new AtomicLong(0); }

  public long get() { return value.get(); }

  public void set(long val) { value.set(val); }

  public void incr() { value.getAndIncrement(); }

}
```

Seems to work…!

Limitations of Atomic variables

Atomic variables only make a class thread-safe only if ONE variable defines the class **state** and there are no **compound actions** accessing this **state**.

- to preserve **state consistency**, you should update related state variables in a **single** atomic operation

Example 2: atomic won't work

```
class BankAccount {
  private int balance = 0;
  int  getBalance(){ return balance; }
  void setBalance(int x) { balance = x; }

// withdraw is a compound action
  void withdraw(int amount) {
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
  }
  … // other operations like deposit, etc.
}
```

Interleaving

Suppose:

- Thread **T1** calls **`x.withdraw(100)`**
- Thread **T2** calls **`y.withdraw(100)`**

If calls interleave

- and **`x`** and **`y`** refer to different accounts…

  -> no data race

- and **`x`** and **`y`** aliases…

  -> possible data race

slide adapted from: Sophomoric Parallelism & Concurrency, Lecture 4

Race condition: A bad interleaving

Interleaved **`withdraw(100)`** calls on the same account

- Assume initial **`balance == 150`**

Thread **T1**

```
int b = getBalance();
```

Thread **T2**

```
int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b – amount);
```

Thread **T1**

```
if(amount > b)
  throw new …;
setBalance(b – amount);
```

Time

balance=50
"Lost withdraw" – unhappy bank

Slide adapted form Sophomoric
Parallelism & Concurrency, Lecture 4

Incorrect "fix"

It is tempting and **almost always wrong** to fix a bad interleaving by rearranging or repeating operations, such as:

```
void withdraw(int amount) {
  if(amount > getBalance())
    throw new WithdrawTooLargeException();
  // maybe balance changed
  setBalance(getBalance() – amount);
}
```

Incorrect “fix”

```
void withdraw(int amount) {
   int b = getBalance();
   if(amount > b)
     throw new WithdrawTooLargeException();
   setBalance(b – amount);
}
```

It is tempting and **almost always wrong** to fix a bad interleaving by rearranging or repeating operations, such as:

```
void withdraw(int amount) {
  if(amount > getBalance())
    throw new WithdrawTooLargeException();
  // maybe balance changed
  setBalance(getBalance() – amount);
}
```

This fixes nothing!

- Narrows the problem by one statement
- (Not even that, since the compiler could turn it back into the old version because you didn’t indicate need to synchronize)
- And now a negative balance is possible – why?

Mutual exclusion is required

- At most one thread withdraws from account **A** at a time
- Exclude other simultaneous operations on **A** too (e.g., deposit)

Mutual exclusion in Java: synchronized methods

A built-in locking mechanism in Java for enforcing **atomicity.**

A method/ function can be made **mutually exclusive** by prefixing the method with the keyword **synchronized**.

```
synchronized void f() { body; }
```

Java implementation

```
class BankAccount {
  private int balance = 0;
  synchronized int getBalance()
    { return balance; }
  synchronized void setBalance(int x)
    { balance = x; }
  synchronized void withdraw(int amount) {
     int b = getBalance();
     if(amount > b)
       throw …
     setBalance(b – amount);
  }
  // deposit would also use synchronized
}
```

Can also be used in example 1: Safe Counter Implementation v2

```
public class Counter {
     private long value;
//[...]
public synchronized void incr() {value++;}
}
```

But atomic class should be faster for this simple example… as avoids actual synchronization behind the scenes and can use hardware support.

Mutual exclusion in Java: synchronised block

Critical sections of code can be made **mutually exclusive** by prefixing the method with the keyword **synchronized**.

- `synchronized void f() { body; }` is equivalent to:
- `void f() { synchronized(this) { body; } }`
- a **synchronized block** has two parts:
  - a reference to an object that will serve as the *lock*
  - a block of code to be guarded by that *lock*

```
synchronized (object) {
statements }
```

How does this work? - Locks

Every object (but not any primitive type) "is a lock" in Java

- A lock is an Abstract Data Type with operations:
  - **`new`**: make a new lock, initially *"not held"*
  - **`acquire`**: blocks if this lock is already currently *"held"*
    - Once *"not held"*, makes lock *"held"*
  - **`release`**: makes this lock *"not held"*
    - if >= 1 threads are blocked on it, exactly 1 will acquire it
- The lock implementation ensures that upon **simultaneous acquires and/or releases**, a correct thing will happen
  - Example: Two acquires: one will "win" and one will block
- How can this be implemented?
  - Need to "check and update" "all-at-once"
  - Uses special hardware and O/S support
    - Covered in operating systems module in CSC3002F

You can use Java's locks directly

```
class BankAccount {
  private int balance = 0;
  private Lock lk = new Lock();
  …
  void withdraw(int amount) {
    lk.acquire(); /* may block */
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
    lk.release();
  }
  // deposit would also acquire/release lk
}
```

Problems with locks

- A lock is a very **primitive mechanism**
  - Still up to you to use correctly to implement critical sections
- **Incorrect:** Use different locks for **`withdraw`** and **`deposit`**
  - Mutual exclusion works only when using **same** lock
- Poor performance: Use same lock for every bank account
  - No simultaneous operations on different accounts
- Incorrect: Forget to release a lock (blocks other threads forever!)
  - Previous slide is wrong because of the exception possibility!

```
if(amount > b) {
  lk.release(); // hard to remember!
  throw new WithdrawTooLargeException();
}
```

Other issues with locks

- If **`withdraw`** and **`deposit`** use the same lock, then simultaneous calls to these methods are properly synchronized
- But what about **`getBalance`** and **`setBalance`**?
  - Assume they're **`public`**, which may be reasonable

- If they don't acquire the same lock, then a race between **`setBalance`** and **`withdraw`** could produce a wrong result

- If they do acquire the same lock, then **`withdraw`** would block forever because it tries to acquire a lock it already has

```
class BankAccount {
  private int balance = 0;
  synchronized int getBalance()
    { return balance; }
  synchronized void setBalance(int x)
    { balance = x; }
   synchronized void withdraw(int amo
      int b = getBalance();
     if(amount > b)
       throw …
     setBalance(b – amount);
  }
  // deposit would also use synchroniz
```

Locks have to be *re-entrant*

```
int setBalance(int x) {
  lk.acquire();
  balance = x;
  lk.release();
}

void withdraw(int amount) {
  lk.acquire();
  …
  setBalance1(b - amount);
  lk.release();
}
```

This simple code works fine provided **`lk`** is a reentrant lock

- Okay to call **`setBalance`** directly
- Okay to call **`withdraw`** (won't block forever)

Re-entrant lock

A re-entrant lock (a.k.a. recursive lock)

- "Remembers"
  - the thread (if any) that currently holds it
  - a *count*

- When the lock goes from *not-held* to *held*, the count is 1

- If (code running in) the current holder calls **`acquire`**:
  - it does not block
  - it increments the count

- On **`release`**:
  - if the count is > 0, the count is decremented
  - if the count is 0, the lock becomes *not-held*

Reentrant locks

- This code would deadlock without the use of reentrant locks:

```
class Widget {
        public synchronized void doSomething()  {
        ....
        }
}

class BobsWidget {
        public synchronized void doSomething()  {
                System.out.println("Calling super");
                super.doSomething();
        }
}
```

Java does it for you: synchronization

Every Java object created, including every Class loaded, has an associated **lock** (or **monitor)**.

- Putting code inside a synchronized block makes the compiler append instructions to **acquire** the lock on the specified object before executing the code, and **release** it afterwards (either because the code finishes normally or abnormally).
- Between acquiring the lock and releasing it, a thread is said to "own" the lock. **At the point of Thread A** wanting to **acquire the lock, if Thread B** already **owns it,** then **Thread A must wait** for **Thread B to release it**.

Java locks summary

Every Java object possesses **one lock**

- Manipulated only via synchronized keyword
- **Class objects** contain a lock used to protect statics
- Scalars like `int` are not Objects, so can only be locked via their enclosing objects

Java locks are **reentrant**

- A thread hitting `synchronized` passes if the lock is free or it already possesses the lock, else waits
- Released after passing as many }'s as {'s for the lock
  - cannot forget to release lock

Who gets the lock?

- There are **no fairness specifications in Java**, so if there is contention to call synchronized methods of an object, an arbitrary process will obtain the lock.

Java: block synchronization versus method synchronization

Block synchronization is preferred over **method synchronization:**

- with block synchronization you only need to lock the critical section of code, instead of whole method.
- Synchronization comes with a performance cost:
  - we should synchronize only part of code which absolutely needs to be synchronized.

Why we need locks

We can't implement our own mutual-exclusion protocol.

```
class BankAccount {
  private int balance = 0;
  private boolean busy = false;
  void withdraw(int amount) {
    while(busy) { /* "spin-wait" */ }
    busy = true;
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
    busy = false;
  }
  // deposit would spin on same boolean
}
```

Still just moved the problem!

busy=false

Time

Thread 1

```
while(busy) { }

busy = true;

int b = getBalance();




if(amount > b)
  throw new …;
setBalance(b – amount);
```

Thread 2

```
while(busy) { }

busy = true;

int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b – amount);
```

“Lost withdraw” – unhappy bank