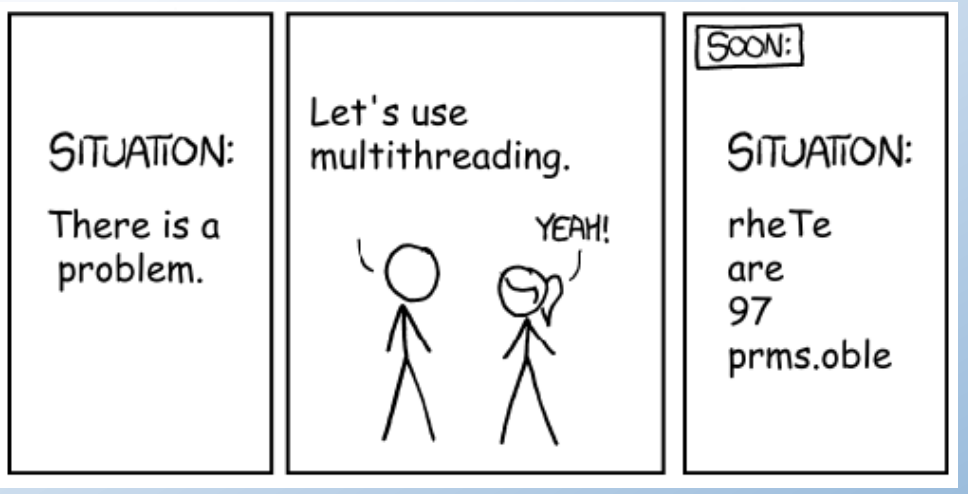
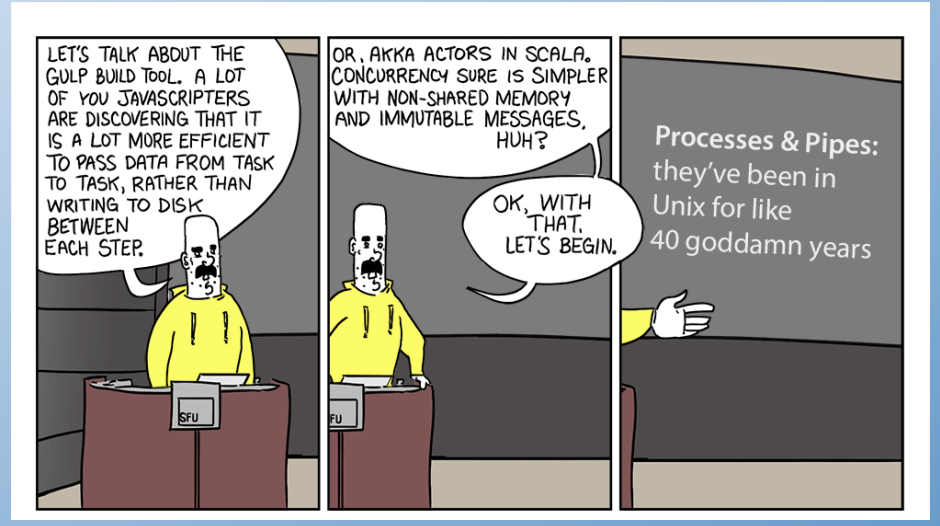
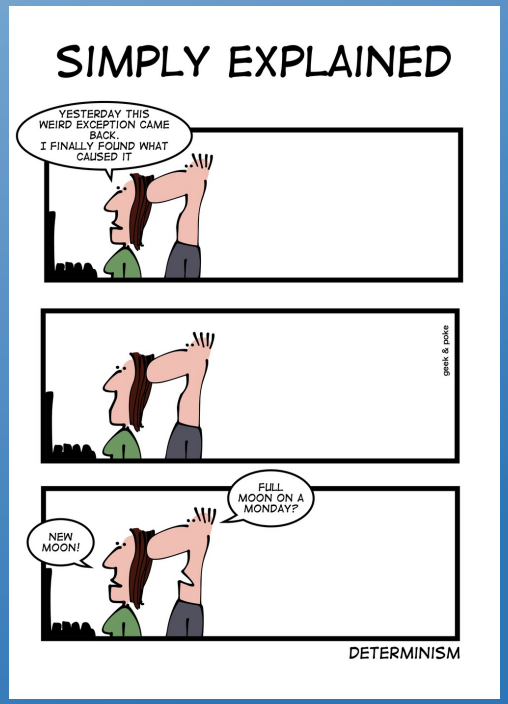


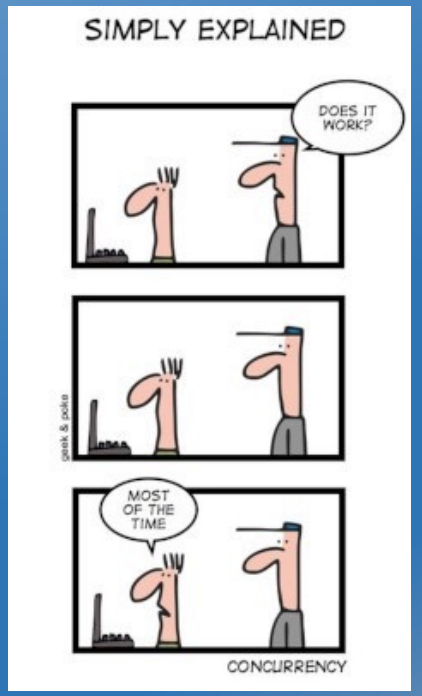


Concurrency and Synchronization

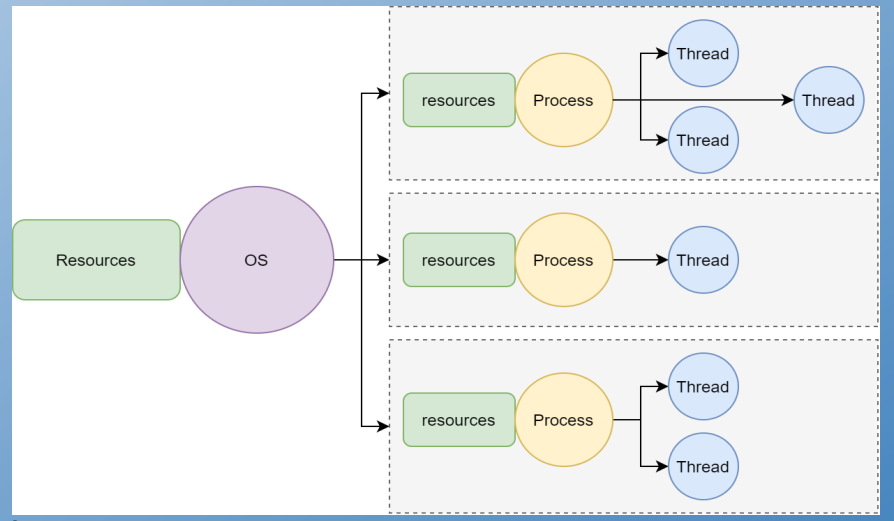
CSCI2340: Software Engineering of Large Systems

Steven P. Reiss





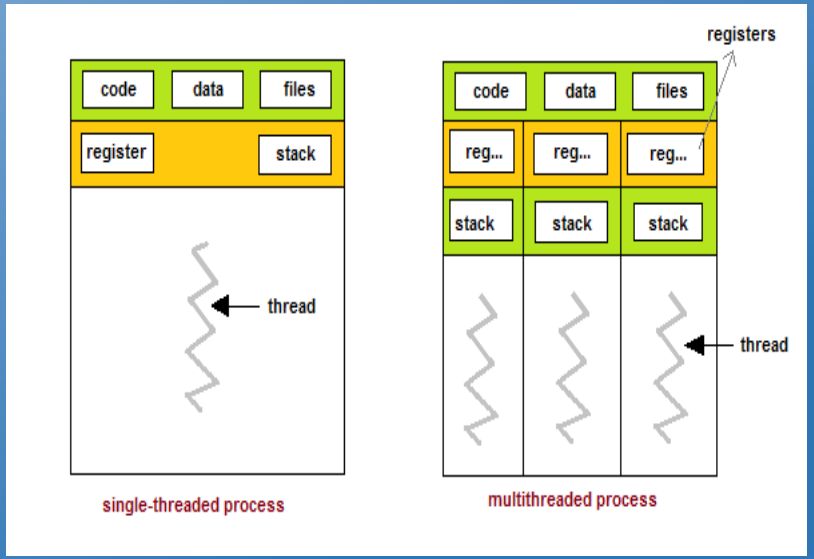
Using Concurrency



- Today's CPUs have multiple cores
  - Can easily run multiple things at once
  - You want your program to take advantage of this
- Doing things in parallel should speed up your program
  - Handle more users, larger data sets, …
- Modern environments require concurrency
  - User interfaces and reactive programming
  - I/O on slow devices such as the web
  - Need to handle large number of simultaneous users
- Most systems today are concurrent
  - All for I/O, many for processing

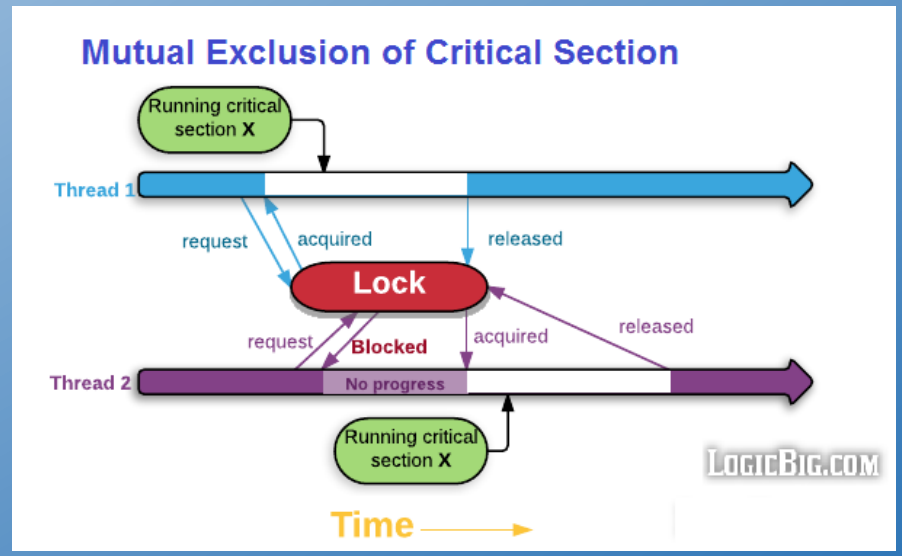
Basic Concepts: Threads

- Threads are schedulable entities that execute code
  - Flow of control
  - Basic way of doing concurrent operations within a process
- Contain:
  - Stack (local variables)
  - Registers (temporary variables)
  - Locks that are held
  - Thread-specific data
- Can be built into the language
  - Java Threads
  - Dart isolates (limited threading – no shared memory)
  - Or built on top (Posix Threads in C; threads in Python)
  - Life is easier if the language knows about threads
- Can be hidden by the language
  - futures; async/await – but these might only provide limited concurrency



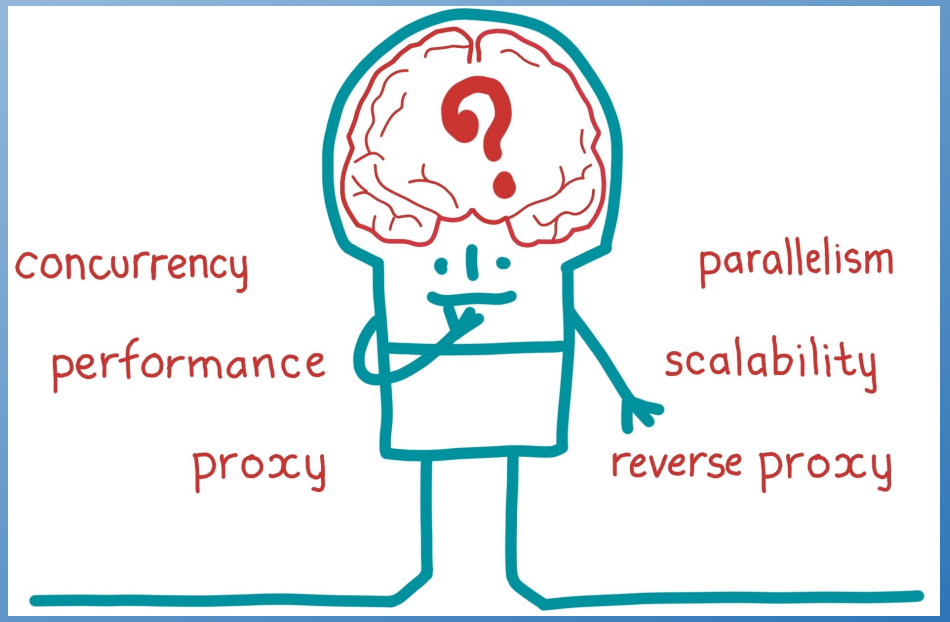
Basic Concepts: Locks

- Locks provide synchronization between threads
  - Ensure that a thread has exclusive use of a resource
  - Access to a shared variable, file, socket, processor
- Different types of locks
  - Mutex, semaphore, monitors, read/write, fork/join
  - Locks on a file; files as a lock
  - Waiting for message replies
  - Java synchronized regions are monitors
    - Wait/notify allows for arbitrary lock definitions
  - Database transactions
  - Hardware locks: Compare and swap
- Concurrent data structures & atomic variables
  - Work without locking; no sequential guarantees
- Locks are needed to make concurrency work
  - Even concurrent data structures use primitive atomic operations

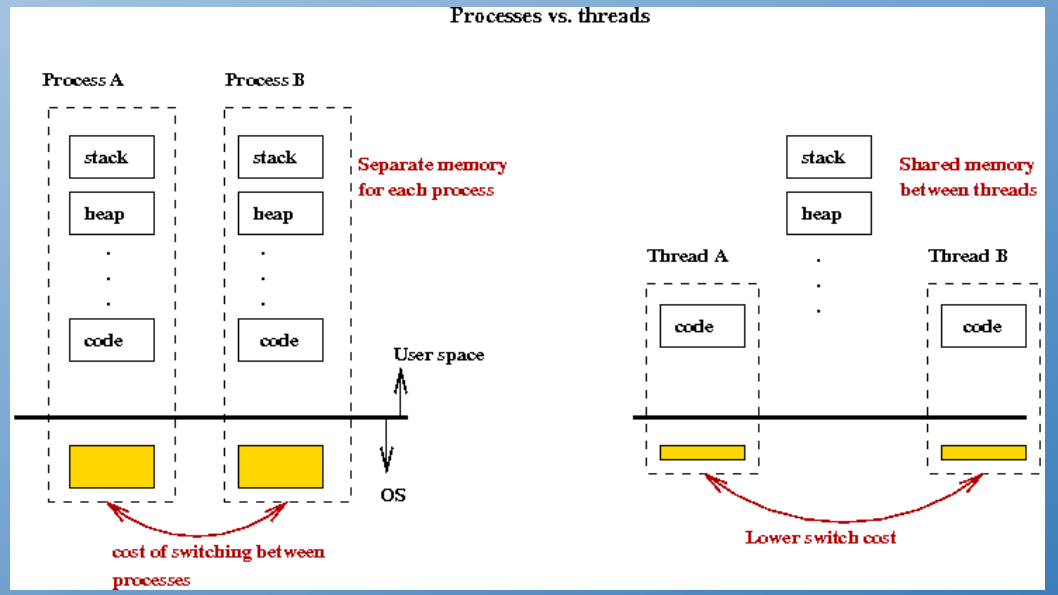


Concurrency and Design

- Techniques discussed earlier
  - Separate processes
  - Within a component
    - Identify potential background tasks
    - Identify what can be done in parallel
    - Identify potential shared data structures
  - User interface concurrency
  - Database concurrency
- Now we need to discuss
  - Concurrency in the detailed design and in the implementation



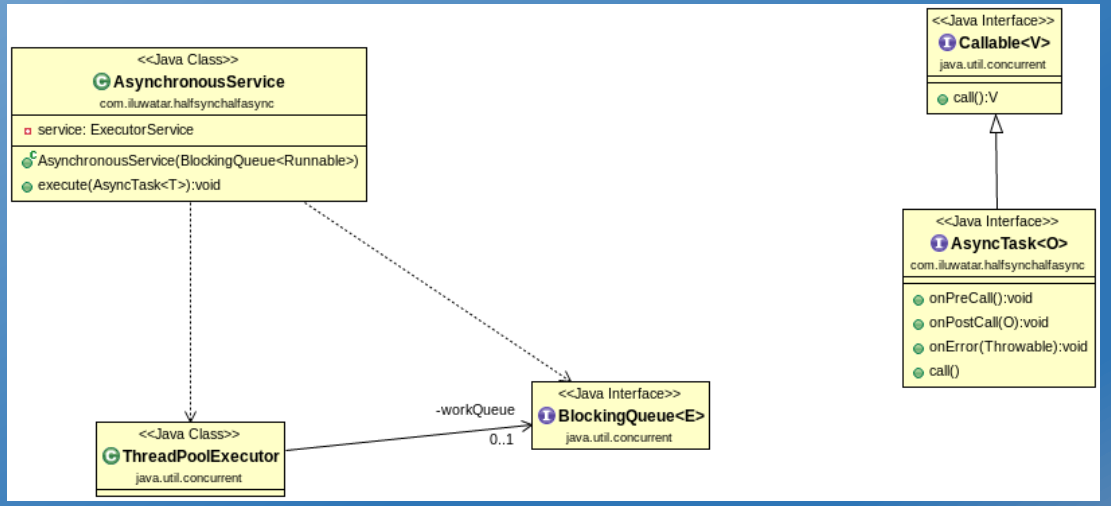
Separate Processes



- Relatively easy to implement
  - Should provide simple concurrency
- Need to handle communication
  - Sharing via messages (HTTP)
  - Sharing via run/exit (waitFor)
  - Sharing via databases, files
- Still need to synchronize
  - Message-based synchronization (wait for a reply)
  - File locks and semaphores (Linux, not Java)
  - Database transactions
- Synchronization problems still exist

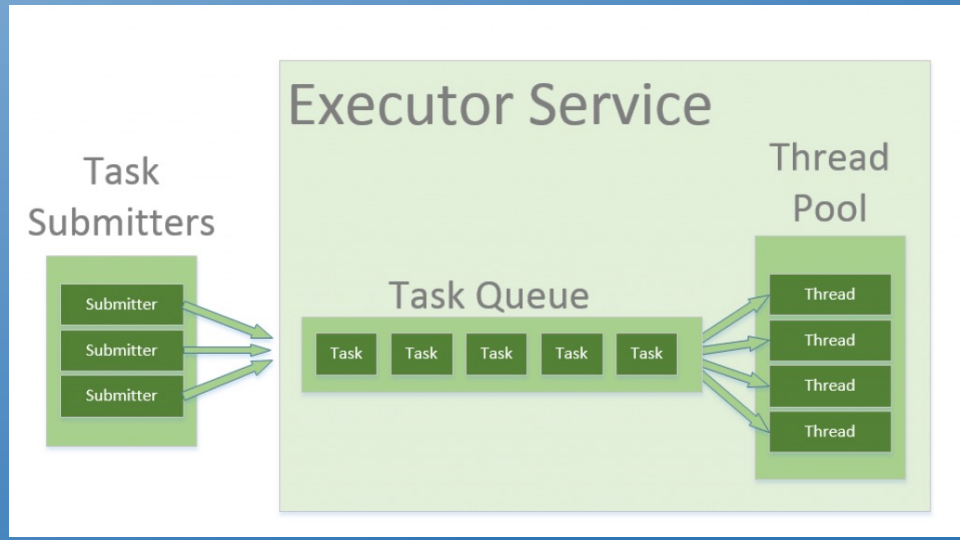
Threads and Parallel (Background) Tasks

- Implement Thread, Runnable or Callable
  - Run method is the background task
  - Runs in its own thread
  - Use Java synchronization to provide a waitFor method
    - Or use java.util.concurrent.Future
- Threads can be run directly
  - Create (new X where X : Thread)
  - x.start()
- Threads are expensive
  - Stack; start-up time; memory



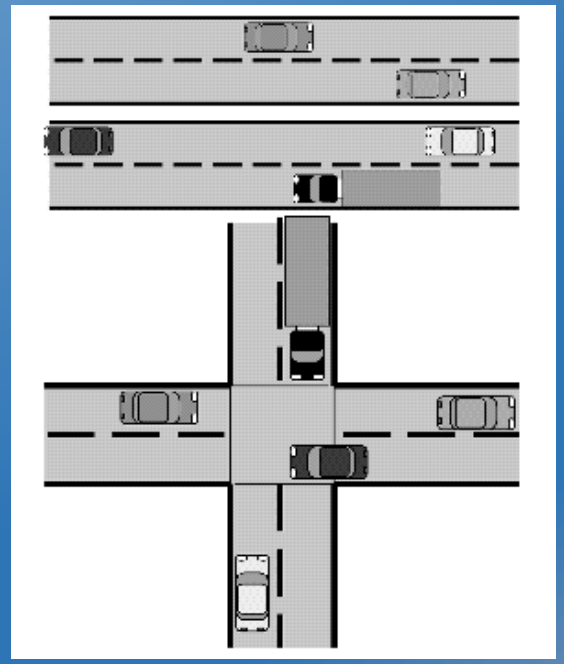
Thread Pools

- Reuse threads for multiple tasks
  - Pool of threads that handle queued tasks
  - Can vary and control the number of threads
  - Can control the queue (e.g., priority queue)
  - Pool can expand and contract dynamically
- Better approach than individual threads
  - But slightly more complex
  - Might need to ensure adequate pool size
  - Can be a little tricky to implement
    - Implementation depends on queue type and size
- But Java thread pools can be tricky to set up
  - Synchronization can become an issue
    - Wait for a task in the queue can fail



Problems with Concurrency

- Concurrency makes programming considerably more difficult
- Synchronization problems: Race Conditions
  - A needs output from B, but doesn't wait for it
  - A and B both want to update a variable at the same time
  - A shouldn't proceed until all Bs are done
- Locking problems
  - Deadlocks: A waits for B, B waits for A
  - Cycles of locks in general
  - Might go beyond explicit locks
- Performance problems
  - You don't get the expected performance boost
  - Parallel programs can even run slower
- Debugging problems



Concurrency Patterns

- Design is all about patterns
  - Standard ways of implementing things
- Patterns for handling concurrency
  - To avoid race conditions
  - To avoid deadlocks
  - To get the maximum performance
- You should be aware of these
  - And use them as appropriate

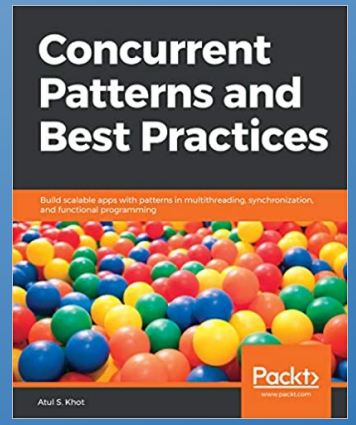
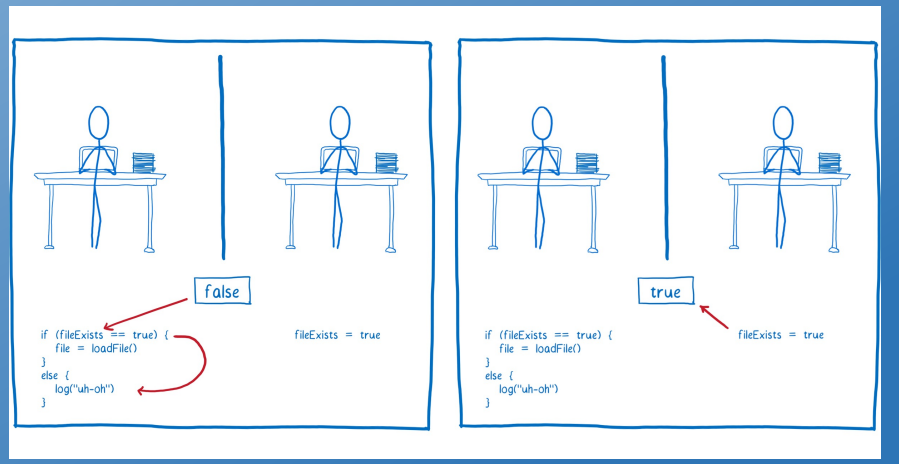


Table 2. Concurrency Design Patterns and Related Quality Attributes

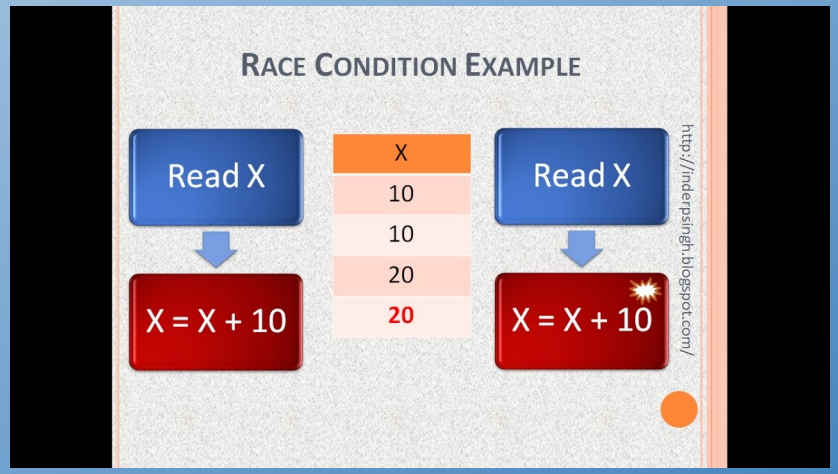
	Quality Attributes vs. Concurrency Design Patterns	Configurability	Flexibility	Interoperability	Modifiability/Maintainability	Modularity	Performance / Efficiency	Portability	Reliability / Robustness (Race Conditions)	Reliability / Robustness (Deadlock)	Reliability / Robustness (Others)	Simplicity	Standards Compliance	Sustainability	Usability
Structural	Double Buffering						•								
	Lock Object								•			•			
	Producer-Consumer		•		•		•								
	Read/Write Lock				•		•		•						
	Scheduler	•			•		•								
	Thread-safe Decorator				•		•			•					
Behavioral	Active Object		•		•		•					•			
	Balking						•				•				
	Event-based Asynchronous				•		•								
	Future				•										
	Guarded Suspension									•					
	Monitor Object								•			•			
	Reactor	•			•	•		•				•			
	Thread-Specific Storage				•		•					•			
Architectural	Asynchronous Processing						•								
	Half-Sync/Half-Async		•		•	•	•					•			
	Leader/Followers				•		•								
Enterprise	AJAX			•	•		•						•		•
	Lock File								•			•		•	
	Optimistic Concurrency						•								
	Session Object				•									•	
	Static Locking Order									•					
	Thread Pool				•		•								
Other General-Purpose	Double-Checked Locking								?						
	Scoped Locking										•				
	Single Threaded Execution								•						
	Strategized Locking		•		•										
	Two-Phase Termination										•				

Race Conditions

- Two threads, A and B, both write to the same location
  - What will another thread see at that location
  - Will it be the value of A or the value of B
    - Or something different
- More typical race conditions
  - Parallel computations
    - A accesses v, computes v', saves v' into v
    - B accesses v, computes v", saves v" into v
  - Thread depends on a variable set in another thread
    - But doesn't wait for that thread to set it



Race Conditions



- Counter Program
  - void count() { ++shared_counter; }
    - Called from multiple threads
  - 4 threads each calling count 100,000,000 times
    - What will the resultant value be
      - Will it be exactly 400,000,000?
      - Can it be over 400,000,000?
      - Can it be under 400,000,000?
- Make a guess

Experiment

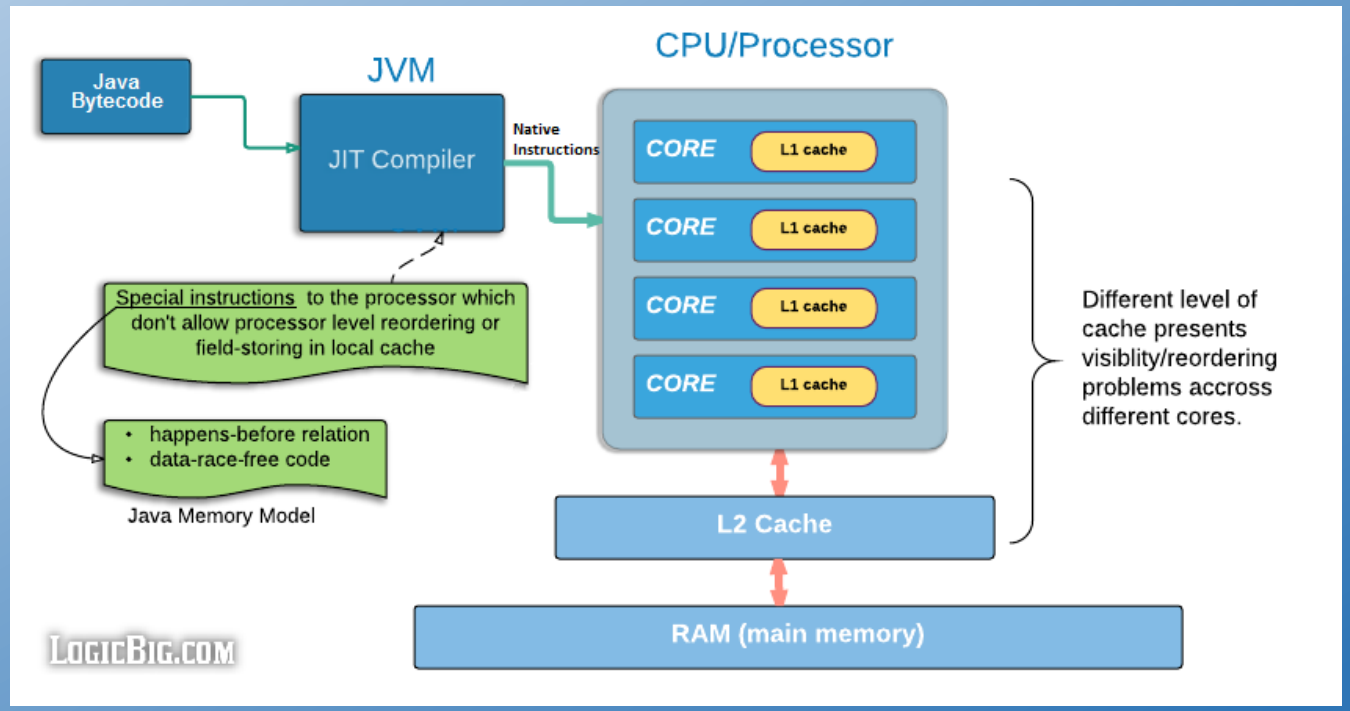
- Tried the counter program as above
- Also tried:
  - Making the count method synchronized
  - Using an atomic integer rather than a simple variable
  - Having each thread maintain its own counter, add these at end
  - Making the shared counter variable volatile
  - Having only one of the threads to the counting
- What results do you expect?
  - Which of these is guaranteed to yield 400,000,000?

Counter Program Results

What	Count	Fred4: clock (cpu)	New Fred4	Mac Mini (intel)
Simple Counter	112,836,399	2.57 (4.25)	0.14 (0.245)	0.16 (0.191)
Synchronized method	400,000,000	19.62 (53.71)	23.06 (80.535)	28.24 (90.31)
Atomic Integer	400,000,000	8.06 (7.22)	3.28 (11.956)	8.36 (32.06)
Separate counter per thread	400,000,000	11.16 (10.96)	2.68 (6.608)	4.08 (8.751)
Simple Counter, volatile	186,035,542	52.80 (14.16)	3.41 (13.449)	9.91 (38.13)
Single Threaded	400,000,000		0.07 (0.073)	0.15 (0.141)

Volatile Variables

- What does volatile mean and why is it used?



Race Conditions

data race

```
// Shared variable
var count = 0

func incrementCount() {
  if count == 0 {
    count ++
  }
}

func main() {
  // Spawn two "threads"
  go incrementCount()
  go incrementCount()
}
```

!data race

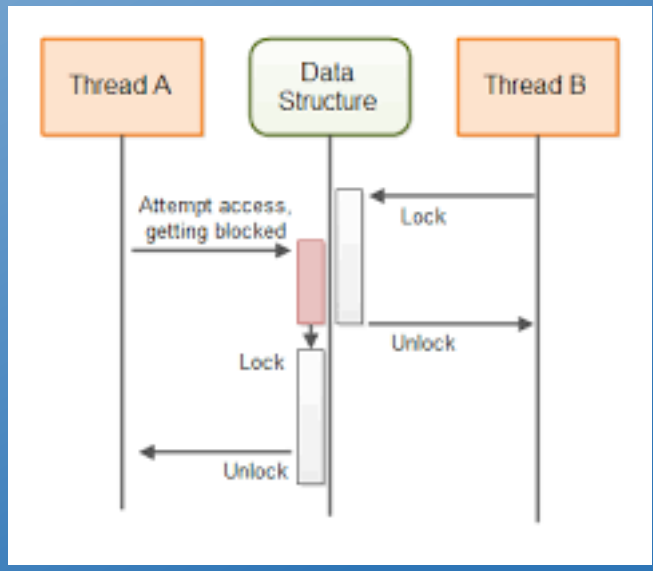
```
Thread 1    Thread 2

lock(l)     lock(l)
count=1     count=2
unlock(l)   unlock(l)
```

- Show up as random results
  - Different results from different runs (with same inputs)
  - Unexpected results
  - Accessing null pointers or undefined values
- Are often difficult to find and understand
  - Debugging these can take months
- Need to avoid these while coding
  - These arise from shared data
  - Identify what classes might be used by multiple threads
    - Or assume that a class is used by multiple threads
  - Identify what data structures might be shared by multiple threads
    - Or assume that a data structure is shared

Design Patterns for Shared Data

- Synchronized data structures in the language
- Concurrent data structures in the language
- Synchronized access to a data structure
- Use immutable objects
- Minimize shared data
- Handling the shared user interface
  - Operations on user interface in the user interface thread

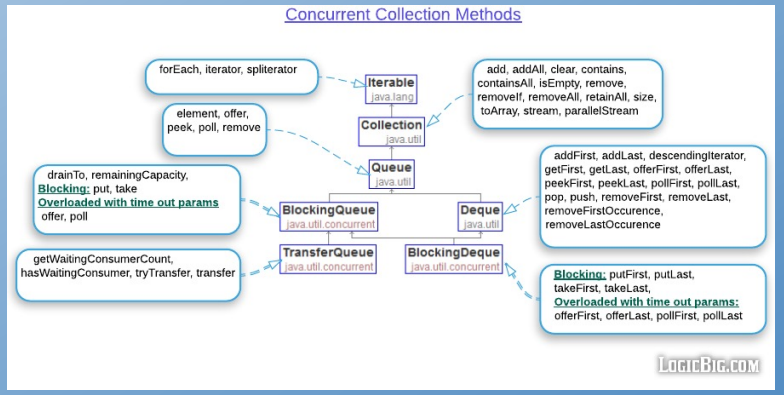


Synchronized Data Structures

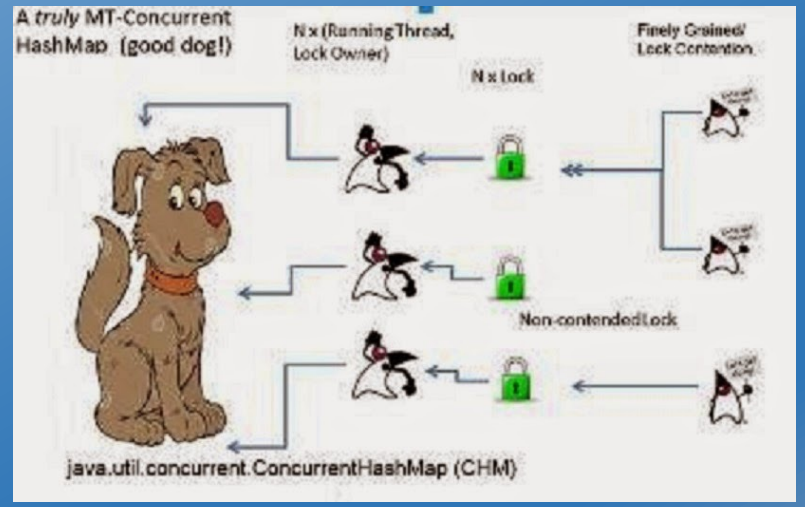
	Synchronized	Thread Safe	Null Keys And Null Values	Performance	Extends	Legacy
HashMap	No	No	Only one null key and multiple null values	Fast	AbstractMap	No
HashTable	Yes	Yes	No	Slow	Dictionary	Yes

- Some Java classes automatically synchronize all access
  - Every method is synchronized
  - Hashtable, Vector, StringBuffer
  - Synchronized collections [Collections.synchronizedSet(…)]
- These don’t always work
  - Common case: add new element if old not present
  - Synchronized requires creating the new element in any case
    - Or synchronized lookup, create, synchronized putIfAbsent, check

Concurrent Data Structures



- Java provides atomic primitives
- Java provides explicitly concurrent data structures
  - ConcurrentHashMap, ConcurrentSkipListSet, ConcurrentSkipListMap, …
  - No guarantee on simultaneous updates and reads
    - May or may not see the updates immediately
    - Simultaneous updates might be in any order
- Encouraged if serializability not needed
  - Can safely iterate over the data
    - But may or may not see newer values
  - Can safely access the data
    - But may or may not see concurrent updates



Synchronized Access to Data Structure

- Use *synchronized (collection) { … }* for all accesses
  - Other than initialization
  - Multiple operations can be combined
  - More general than synchronized data structures
  - Might need to use it for reads as well as writes
  - Will need to use for iterations
- Caveats
  - All updates to a data structure should be in a single class
    - Generally, return a read-only copy of the collection if others want to scan it
  - Updates should be localized (few update methods)
  - Be careful of creating potential deadlocks
  - Easy to miss one or more locations

Type	How To Synchronize?
ArrayList	Collections.synchronizedList()
HashSet	Collections.synchronizedSet()
HashMap	Collections.synchronizedMap()

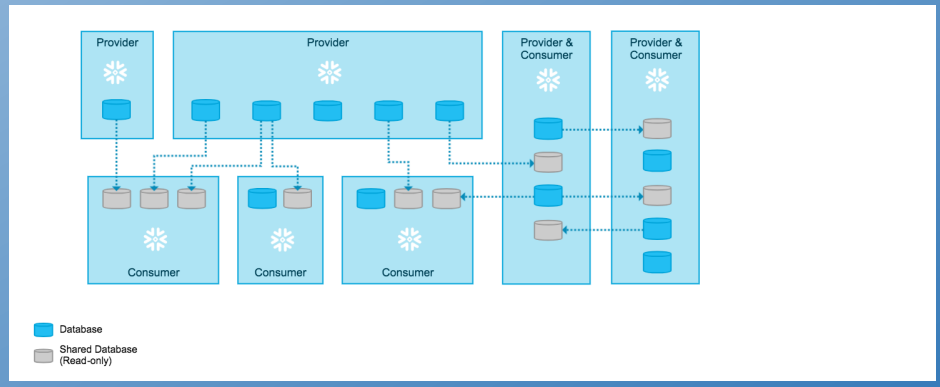
Immutable Objects

- Immutable Objects
  - Created once, never modified
- These are inherently thread-safe
- Might want unique immutable objects
  - Requires synchronization on creation, not on use

```
108  * @since   JDK1.0
109  */
110
111 public final class String
112     implements java.io.Serializable, Comparable<String>, CharSequence {
113     /** The value is used for character storage. */
114     private final char value[];
115
116     /** Cache the hash code for the string */
117     private int hash; // Default to 0
118
119     /** use serialVersionUID from JDK 1.0.2 for interoperability */
120     private static final long serialVersionUID = -6849794470754667710L;
121
122     /**
123      * Class String is special cased within the Serialization Stream Protocol.
124      *
125      * A String instance is written into an ObjectOutputStream according to
126      * <a href="{@docRoot}/../platform/serialization/spec/output.html">
127      * Object Serialization Specification, Section 6.2, "Stream Elements"</a>
128      */
129     private static final ObjectStreamField[] serialPersistentFields =
130         new ObjectStreamField[0];
131
132     /**
133      * Initializes a newly created {@code String} object so that it represents
134      * an empty character sequence.  Note that use of this constructor is
135      * unnecessary since Strings are immutable.
136      */
137     public String() {
138         this.value = "".value;
139     }
140
```

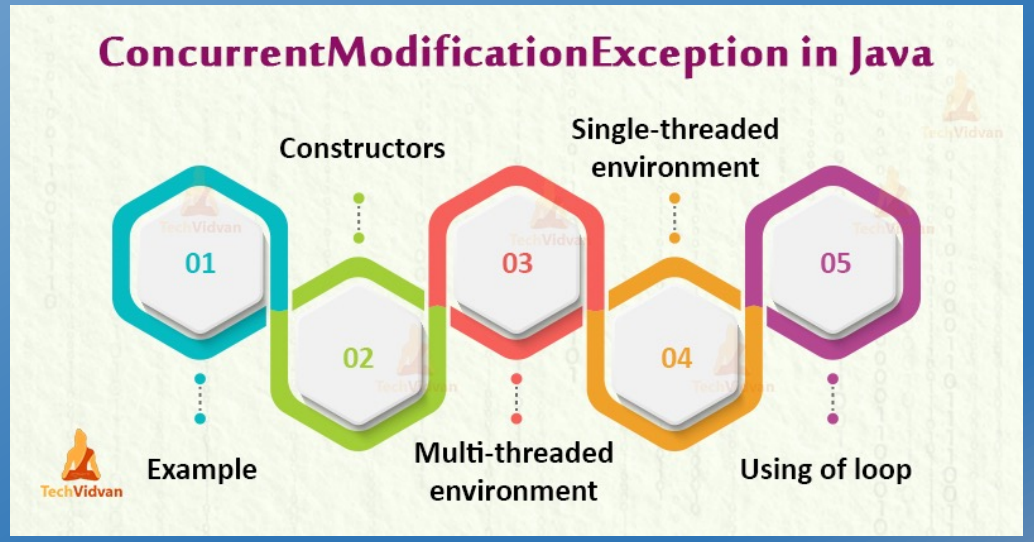
Minimize Shared Writable Data

- Read-only data is easily shared
  - Constructors don't need synchronization
  - Or when you don't worry about updates
    - Ball position for example
- Identify what needs to be shared
  - Now and possibly in the future
- If you don't know about a class
  - Assume it will be eventually shared and code accordingly
  - If not coded for multiple threads, document as not thread safe

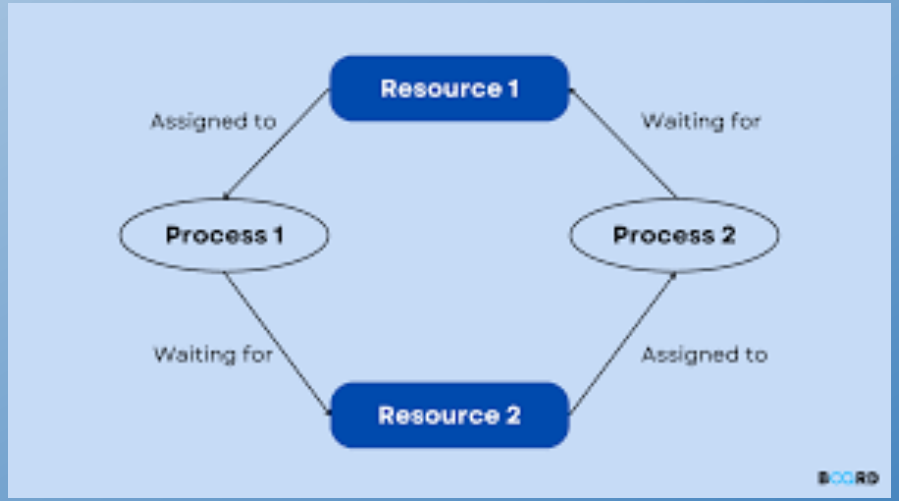


ConcurrentModificationException

- Another form of race condition
  - But it can happen even without threading
- Looping over a structure that is changed
  - foreach loops
  - Even non-loops [e.g., new ArrayList<>(structure)]
- Patterns to avoid this
  - Create copy to iterate over
    - Synchronize the creation if needed
  - Use iterators & their operators
    - If single threaded
  - Use concurrent data structures
  - Synchronize the loop (NO)

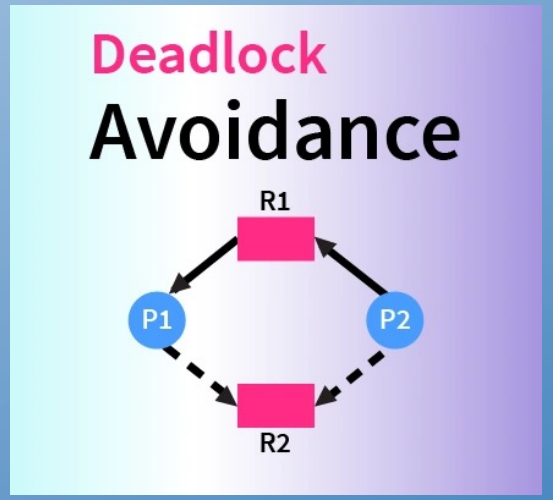


Deadlocks



- Threads waiting on each other
  - No thread can make any progress
    - Thread A locks X, tries to lock Y
    - Thread B locks Y, tries to lock X
  - Can be arbitrary cycles (more than 2 locks)
  - Can involve different types of locks
  - Can involve system locks as well as user locks (User Interface)
  - Can involve waiting for messages between processes
  - Can involve waiting for your turn in a thread pool
- Evolution and Maintenance
  - Tends to add locks to avoid race conditions
  - Adding locks tends to create dead locks

Patterns to Avoid Deadlocks



- Locks should be as local as possible
  - Use synchronized statements rather than methods
  - Avoid calling other routines from within a locked region
    - Especially routines that might lock something else
  - Include locking in documentation for method
  - Avoid executing long-running or non-trivial code in a locked region
- Avoid waiting indefinitely for a remote process or message
  - These are effectively locks & remote process might fail
  - Also avoid waiting on message before producing a response
- Avoid having lots of threads waiting on a single lock
  - This can cause problems with thread pools
  - Can be difficult with hidden locks

More Patterns to Avoid Deadlocks

- Use synchronized statements rather than methods
  - To minimize the code that is synchronized
- Use java.util.concurrent locks
  - More control (e.g., read-write locks, semaphores, trylock)
  - More difficult to code with (use try…finally)
  - Avoids bad code inside a monitor
  - This doesn't avoid deadlocks
- Split locked regions into smaller regions
  - Synchronized check
    - Can this be unsynchronized?
  - Unsynchronized work
  - Synchronized ***recheck*** and save
- Avoid nested locks
  - Or document and use a fixed lock ordering
- Avoid locking where possible: use concurrent data structures

Case 1

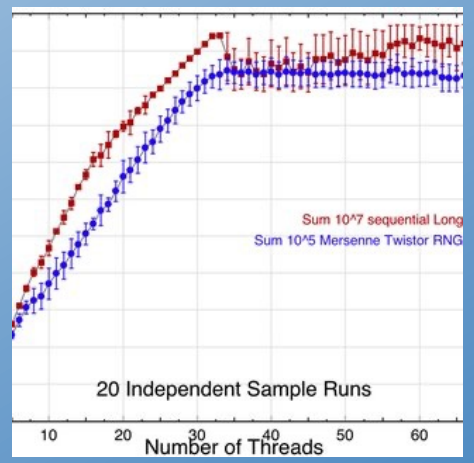
```
thread-1                    thread-2
deposit(int val){           deposit(int val){
   int tmp = bal;              int tmp = bal;
   tmp = tmp + val;            tmp = tmp + val;
   bal = tmp;                  bal = tmp;
}                           }
```

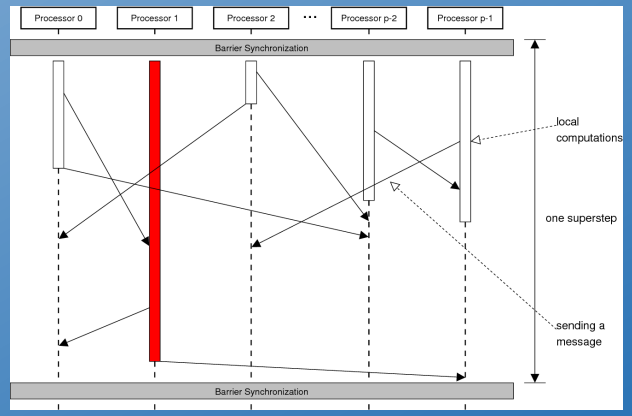
Case 2

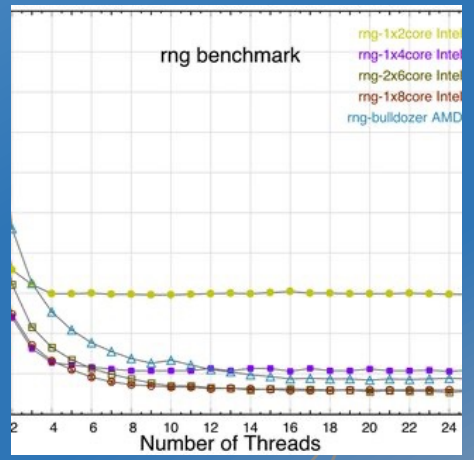
```
thread-1                    thread-2
deposit(int val){           deposit(int val){
   synchronized(o){            synchronized(o){
     int tmp = bal;              int tmp = bal;
     tmp = tmp + val;            tmp = tmp + val;
   }                           }
   synchronized(o){            synchronized(o){
     bal = tmp;                  bal = tmp;
   }                           }
}                           }
```

Performance Problems

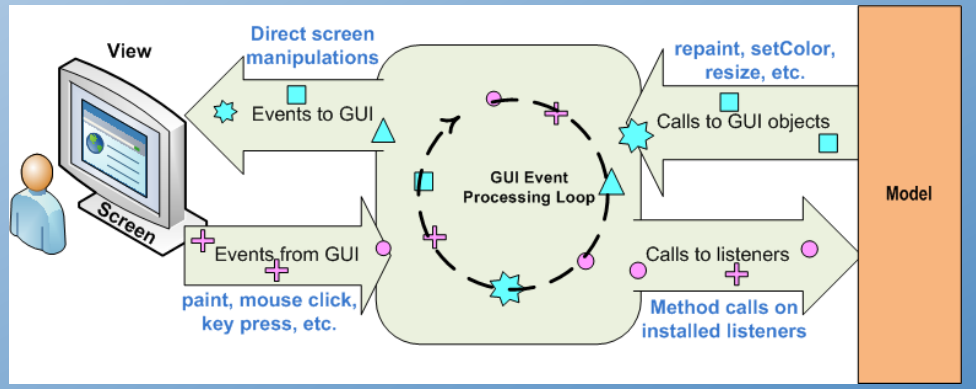
- Synchronization can be expensive
  - Counter program with & without synchronization
  - Cost of locking even when not needed
  - Cost of volatile variables
- You never get as much as you expect
  - Can be slower with multiple threads than single thread
  - Speed up degrades with # of threads
  - Speed up depends on the problem involved
  - Speed up depends on hardware involved





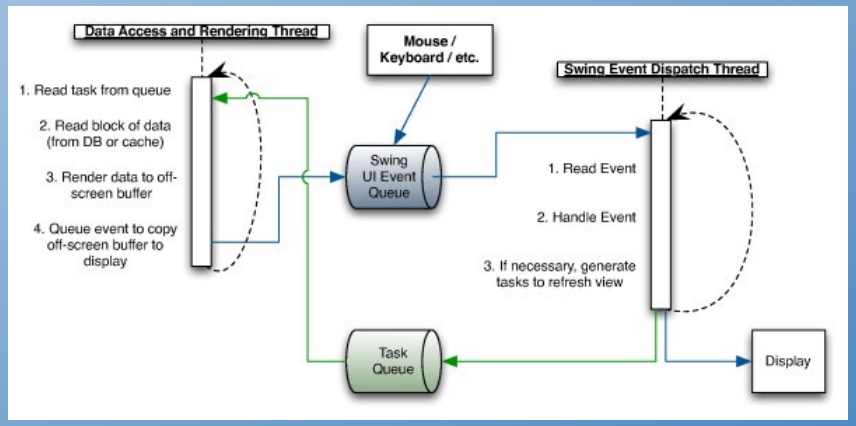


User Interface Threads



- The user interface runs in its own thread
  - Can access all widgets, definitions, etc.
  - These are effectively shared
- All changes to the UI should be done in UI thread
  - After initialization
  - Creating new windows, updating widget properties
  - Updating tree, list, table contents
    - This is why there are table models independent of the display
  - Updating selection, cursor, …
- Use SwingUtilities.invokeLater(…) or invokeAndWait

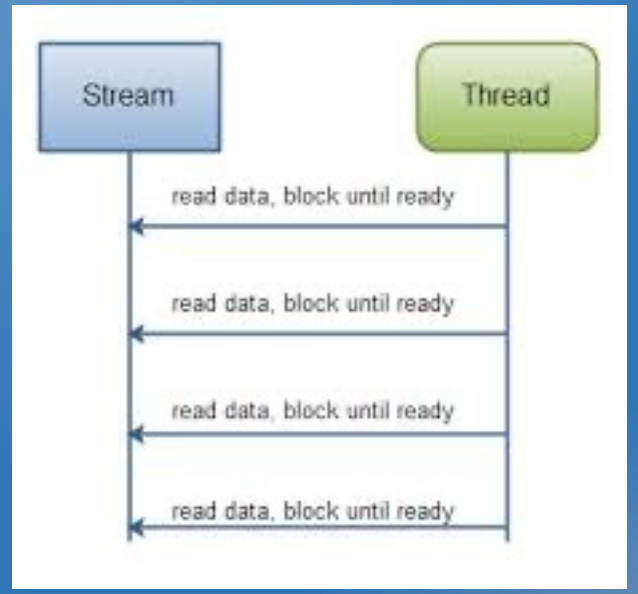
User Interface Thread



- User interface callbacks are in UI thread
  - User interface invokes user routine in that thread
  - Anything done in the callback blocks the UI
- All non-trivial UI actions should be runnable tasks (> 10ms)
  - Done in a separate thread (with a thread pool)
  - Using SwingUtilities.invokeLater to update UI if needed
- The complexity of callbacks will grow over time
  - What is simple now might be time consuming in the future
- JavaScript, Dart
  - All futures, async calls are done in the UI thread
    - But I/O and external processes are done in parallel with it
  - Long-running computations
    - Either need to be put in a separate process
    - Or need to be split up into small steps

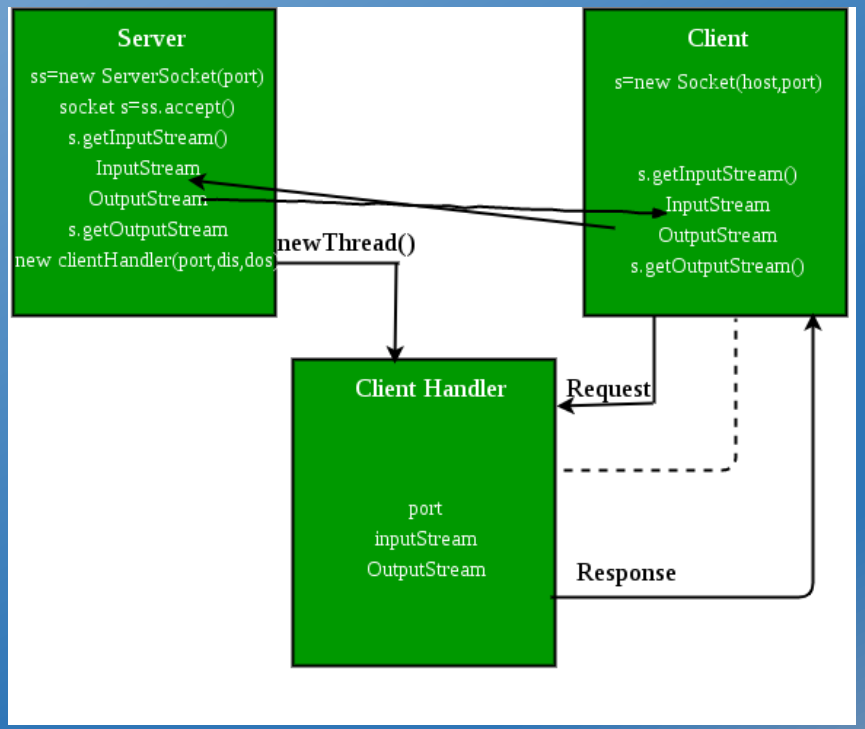
Input / Output Concurrency

- I/O is slow with respect to computation
  - Don't want to slow program down while doing I/O
- Threads provide a convenient way to handle overlap
  - I/O often needs to wait
  - Threads provide a simple mechanism to allow this
    - Make non-blocking I/O easy to program
    - Thread waits for the I/O operation
    - The rest of the program continues (in other threads)
  - Thread overhead is small compared to I/O



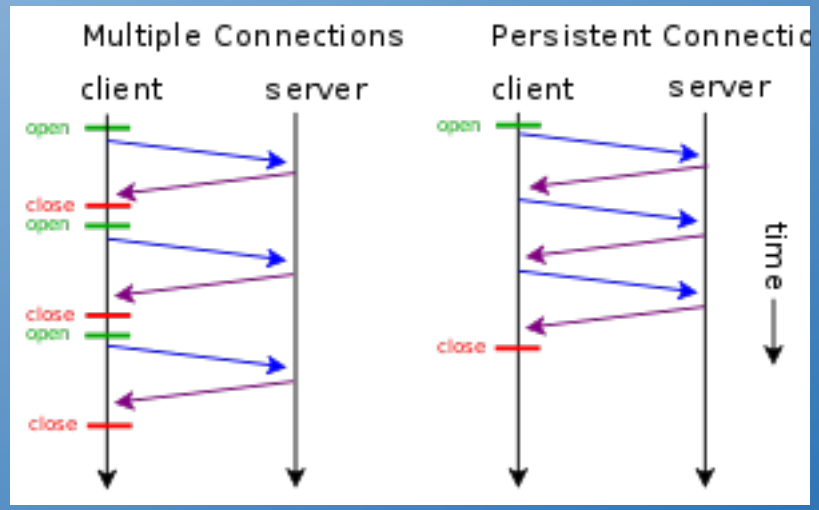
Sockets and Threads

- Server Socket Thread
  - Implemented as a thread that opens server socket
  - Loops forever doing accepts
  - Creates new client thread for each accept
- Client Socket Thread
  - Given socket for client
  - Loops reading from client
    - When a full message is read, process the message
    - Send the reply
  - Exits when socket is closed
  - If two-way, provide call to send a message
    - Need to handle replies – gets more complex

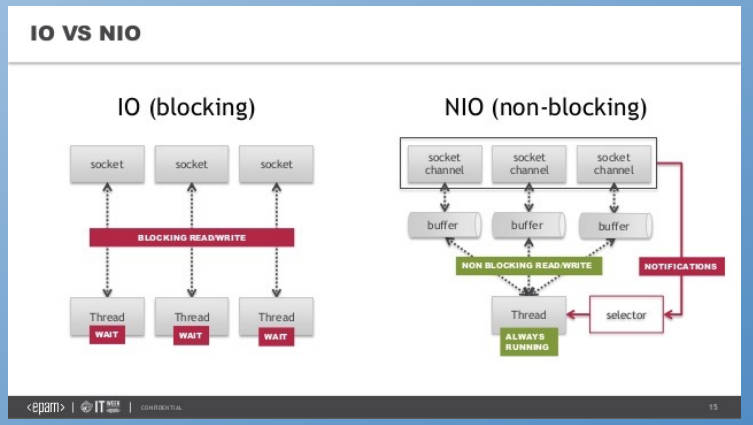


I/O Threads for Other Uses

- Connecting to external server
  - HTTP connection
  - Stdin, stdout, stderr from the subprocess
  - Separate thread that reads and reacts
  - Separate thread with futures
- Large disk I/O operations
  - Can also be done in background
- Waiting for a message
- Waiting for external process to terminate



Threaded I/O Alternatives

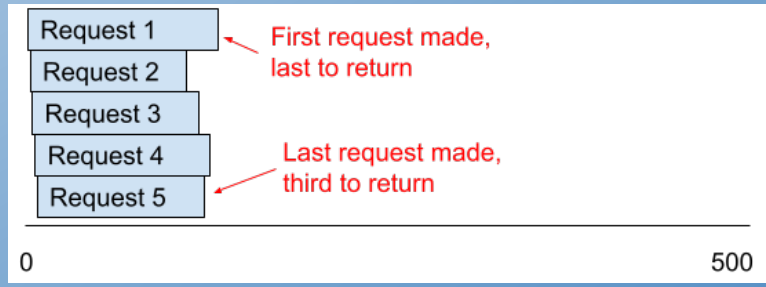


- Use non-blocking I/O operations
  - C/C++ use interrupts or exceptions on completion
  - Java: (java.nio) implemented using futures
- These can be a bit more complex to understand
  - And to get proper overlap
  - Non-threaded languages use these internally
    - Providing futures or async calls for I/O

Implicit Concurrency

- JavaScript, Dart, and other languages
  - async - await, futures
- Concurrency for external actions
  - I/O and external actions done in background (HTTP requests, Database requests)
  - Libraries provided to do complex actions in background
    - Either as language extensions or as plug-ins
  - Running your own server programs
  - Assumption made that these are where your program is spending its time
    - This is true for most web servers, web pages mobile apps
- But your code in JavaScript or Dart is still single-threaded
  - One event loop that runs one thing at a time
  - Need to move intense computations to a separate process
    - Or do low-level coding to handle these in the language
  - Dart has isolates, but these can be tricky to use (no shared memory)
    - Good for handling separate users in separate threads

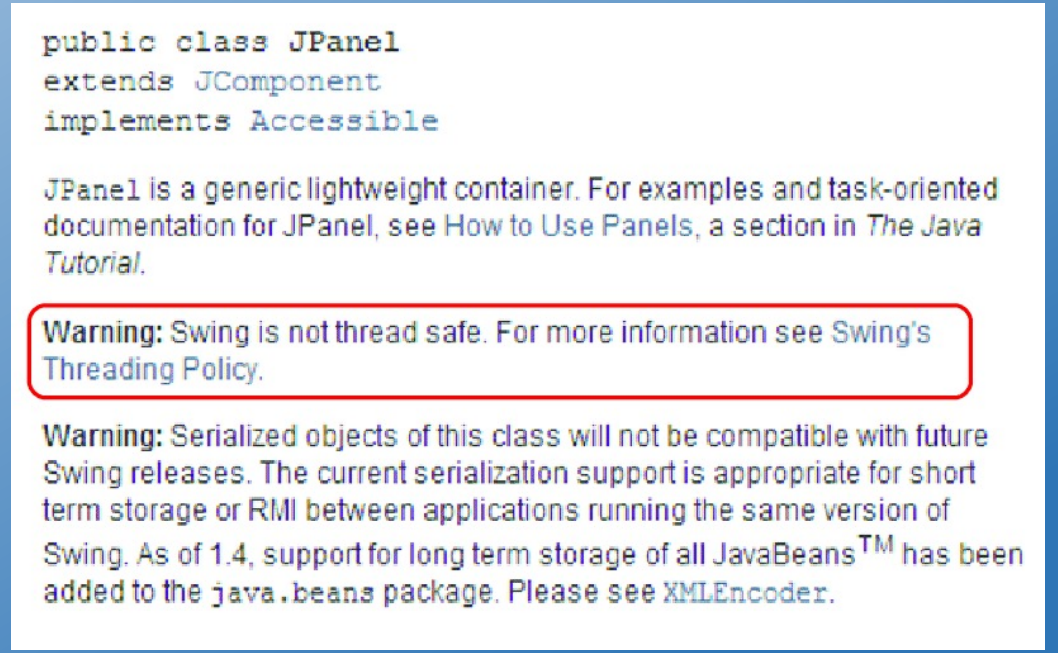
Implicit Concurrency Problems



- There can still be concurrency problems (but not as many)
  - Race conditions with database operations (use transactions)
    - Handling failed transactions
  - Race conditions on read – await – compute based on read – write
  - Deadlocks with external processes and messages
    - Handling failed messages
    - Ensuring external failures are noticed by the program
- Performance
  - Don’t want to block the main event loop with computation
  - Anything complex or time consuming needs to be external
  - Or requires low-level coding or server processes

Documentation for Concurrency

- Document each class or module
  - Immutable
  - Thread-safe
  - Not thread safe
- Document methods
  - What they lock



```
public class JPanel
extends JComponent
implements Accessible
```

JPanel is a generic lightweight container. For examples and task-oriented documentation for JPanel, see How to Use Panels, a section in *The Java Tutorial*.

Warning: Swing is not thread safe. For more information see Swing's Threading Policy.

Warning: Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications running the same version of Swing. As of 1.4, support for long term storage of all JavaBeans™ has been added to the `java.beans` package. Please see `XMLEncoder`.

HOMEWORK

- Its time to recode your programming assignment
  - It turns out that compiling on my new Linux machine takes < 10 seconds
- Various alternatives:
  - Use your bouncing balls for data visualization
    - Show information about all processes or your processes as you thought about earlier
    - Show information about weather in 100+ cities (specify in a file)
    - Show information about 100+ stocks (specify in a file)
    - Show information about some other time-changing values
  - If your program has a goal the user should achieve
    - Then create an AI agent that tries to achieve the same goal
    - AI agent should run independently from the ball computations
  - If your program does complex computations (e.g., collisions or inter-ball gravity)
    - Then code the program to do these in separate threads (using a thread pool?)
    - And experiment to see how many threads provide the best performance
    - Provide graphs and discussion of how the number of balls and number of threads affects performance
- Also consider updating the program
  - Recode your assignment to use concurrency (needed with any of the above options)
  - Run in an elliptical or polygonal window
- Due 10/24 (can get it done earlier to provide time on project)

PROJECT HOMEWORK

- Continue detailed design of your project piece
  - You should have a sense of the top-level classes
    - Of your component
- Hand in individual top-level designs next Tuesday (10/22)
- Should have time for a project meeting on Thursday