50.003: Elements of Software Construction

Week 9

Concurrency: Performance

Use An Existing Library If You Can.

Concurrency-Related Library

- Synchronized Collections
- Concurrent Collections
- Synchronizers (next week)

Synchronized Collections

- Some of the older (jdk2) collections are synchronized and thus thread-safe.
  - Vector:
  - Hashtable
  - Collections.synchronizedXxx
- These are essentially built with synchronized methods.

Synchronized Classes

- Why is this not ideal?

```
public static void doSomething (Vector list) {
      synchronized (list) {
            for (int i = 0; i < list.size(); i++) {
                  doSomething(list.get(i));
            }
      }
}
```

- It is not efficient if list is big and/or doSomething() is slow.

Vector is considered a “legacy” collection class. “Modern” collection Classes use Iterator.

Example

- What could happen to the following code if widgetList is accessed by multiple threads?

```
List<Widge> widgeList = Collections.synchronizedList(new  ArrayList<Widget>());
…
for (Widget w : widgetList) {
		doSomething(w);
}
```

***Internally, javac generates code that uses an Iterator, repeatedly calling hasNext and next to iterate the list. If a modification count (associated with the collection) changes during iteration, hasNext or next throws ConcurrentModificationException. This is called fail-fast.

Hidden Iterator

- Lock everywhere a shared collection might be iterated.

```
class HiddenIterator {
        private final Set<Integer> set = new HashSet<Integer>();

        public synchronized void add(Integer i) { set.add(i); }
        public synchronized void remove (Integer i) { set.remove(i); }

        public void addTenThings() {
                Random r = new Random();
                for (int i = 0; i < 10; i++) {
                                add(r.nextInt());
                }
                System.out.println ("DEBUG: added ten elements to " + set);
        }
}
```

While locking can prevent iterators from throwing `ConcurrentModificationException`, you have to remember to use locking everywhere a shared collection might be iterated. This is trickier than it sounds, as iterators are sometimes hidden. There is no explicit iteration in `HiddenIterator`, but the code in bold entails iteration just the same. The string concatenation gets turned by the compiler into a call to `StringBuilder.append(Object)`, which in turn invokes the collection's `toString` method - and the implementation of `toString` in the standard collections iterates the collection and calls `toString` on each element to produce a nicely formatted representation of the collection's contents.

The real lesson here is that the greater the distance between the state and the synchronization that guards it, the more likely that someone will forget to use proper synchronization when accessing that state. If `HiddenIterator` wrapped the `HashSet` with a `synchronizedSet`, encapsulating the synchronization, this sort of error would not occur.

Iteration is also indirectly invoked by the collection's `hashCode` and `equals` methods, which may be called if the collection is used as an element or key of another collection. Similarly, the `containsAll`, `removeAll`, and `retainAll` methods, as well as the constructors that take collections are arguments, also iterate the collection. All of these indirect uses of iteration can cause `ConcurrentModificationException`.

Discussion

- Problems with locking a collection
  - If the collection is large or the task performed is lengthy, other threads could wait a long time.
  - It increases the risk of problems like deadlock.
  - The longer a lock is held, the more likely it is to be contended.
- Alternative?
  - Clone the collection, lock and iterate the copy.

Concurrent Collections

- Java 5.0 improves on the synchronized collection by providing several concurrent collection classes.
- Replacing synchronized collections with concurrent collections can offer dramatic scalability improvement with little risk.

ConcurrentHashMap

- It is a HashMap designed for concurrency.
- It uses a finer-grained locking mechanism called lock striping.
  - Uses an array of 16 locks.
  - Each lock guards 1/16 of the hash buckets.
  - Bucket N is guarded by lock N mod 16.
- The iterators returned by ConcurrentHashMap are weakly consistent (i.e., it is OK to modify the collection while iterating through it) instead of fail-fast.
- It does not support client-based locking.

weakly consistent -> when you are modifying this part of data structure, some other thread can modify other part of the data structure.
client-based locking: 16 locks are private objects in ConcurrentHashMap class itself. A client cannot access these locks, so it does not support client-based locking.
You cant lock based on the class of ConcurrentHashMap. (i.e. using *this*)

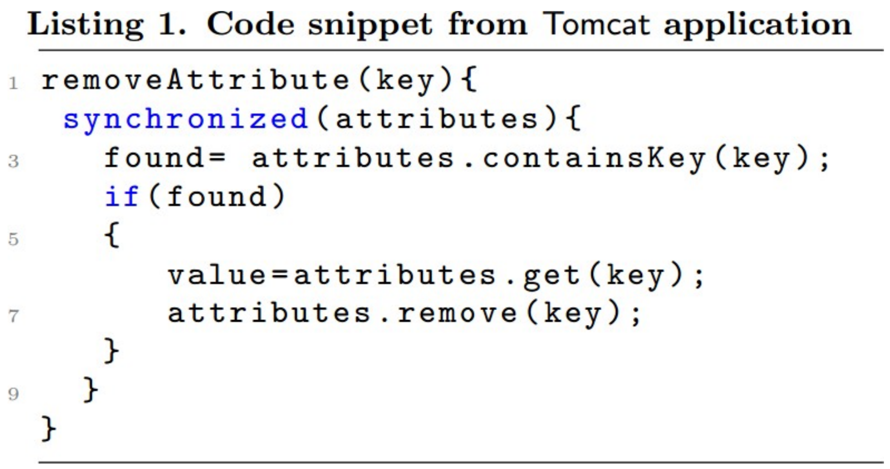
doing client-based locking on ConcurrentHashMap
Synchronising on attributes does not prevent other threads from modifying the hashmap. (by calling other methods in ConcurrentHashMap class)
so found can be false but after that the key is entered.

One Case

- Tomcat Bug: 53498

- The fix (attributes is a ConcurrentHashMap object)

Client side locking *- Classes like ConcurrentHashMap doesn't support Client side locking because get method is not using any kind of lock. so although you put a lock over its object like synchronized (object of ConcurrentHashMap) still some other thread can access object of ConcurrentHashMap.*


Listing 1. Code snippet from Tomcat application

```
removeAttribute(key){
 synchronized(attributes){
   found= attributes.containsKey(key);
   if(found)
   {
      value=attributes.get(key);
      attributes.remove(key);
   }
  }
}
```

CopyOnWriteArrayList

- It is a concurrent replacement for a synchronized list that offers better concurrency in some common situations.
- A new copy of the collection is created and published every time it is modified.
- All write operations are protected by the same lock and read operations are not protected.

CopyOnWriteArrayList

- http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/4797cd0713b4/src/share/classes/java/util/concurrent/CopyOnWriteArrayList.java
- Note: line 99, line 405, line 112
- When is this probably not efficient?

if most cases are writing. Then overhead is copy

BlockingQueue

- Producer-Consumer patterns are very common in programs; Usually some kind of buffering is involved between P and C; The buffer can be implemented as a blocking queue.
- Blocking queues are a powerful resource management tool for building reliable applications: they make your program more robust to overload by throttling activities that threaten to produce more work than can handled.

Cohort Exercise 1

- Analyze GDesktopProb.java which implements the skeleton of Google Desktop. There is some potential problem regarding thread-safety of the implementation. Fix it.

FutureTask

- Future task are made up of Future and Callable, the result bearing relative of Runnable.

Sample program: FutureTaskExample.java

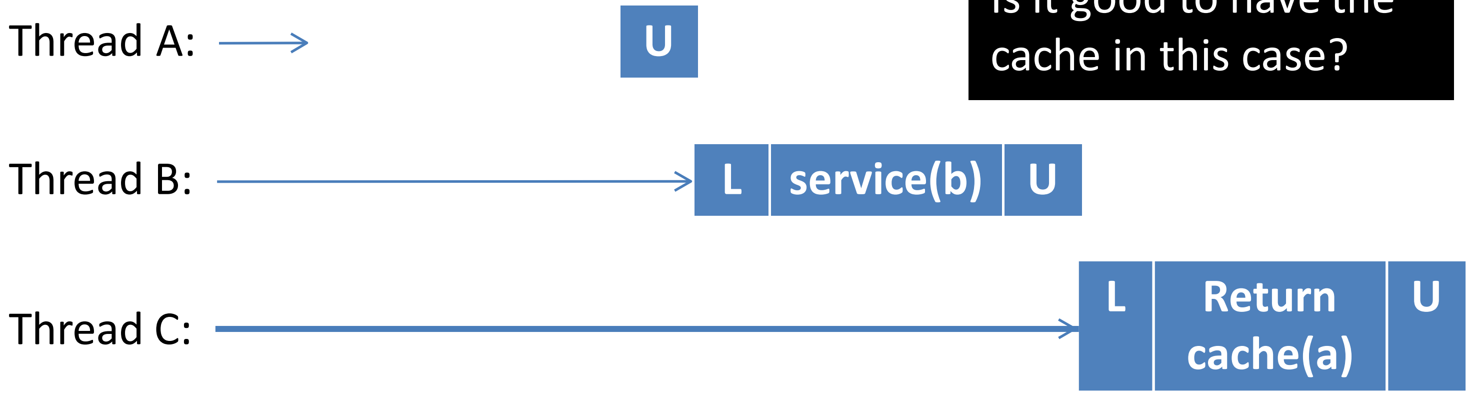
- There are several ways to complete:
  - Normal completion; cancellation; and exception.
- Once a FutureTask is completed it cannot be restarted.
- Future.get() returns the result immediately if ‘the future is here’ (Task is completed) and Blocks if the task is not complete yet.

Example

- Nearly every server application uses some form of caching.

Click here for a sample program: Cache.java

- Poor concurrency



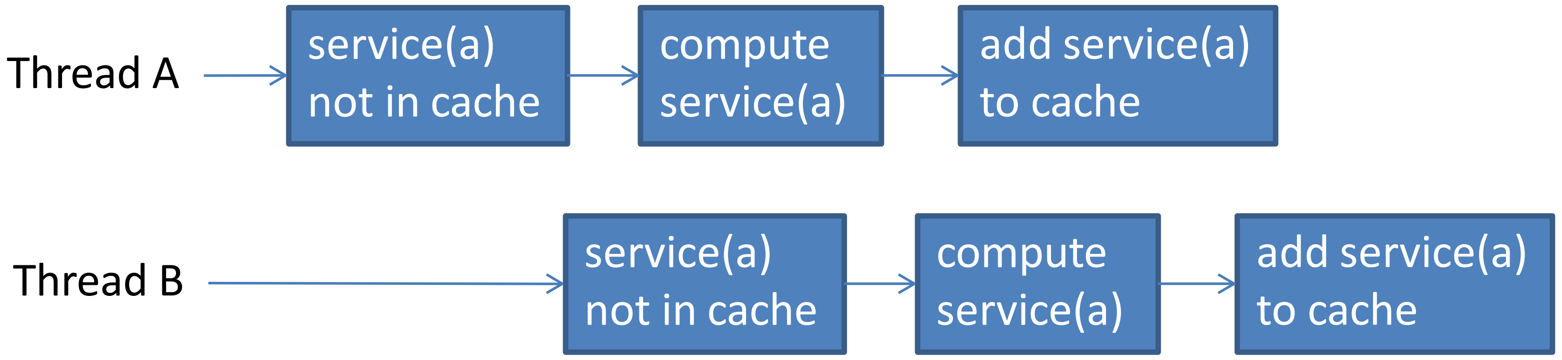
Is it good to have the cache in this case?

Click here for a sample program: CacheV1.java

Example

if 2 threads come in with same number:
thread 1: *not in ConcurrentHashMap*
factor the number
thread 2: *same number but factoring from thread 1 not done yet*
start factoring the same number

- Minor problem

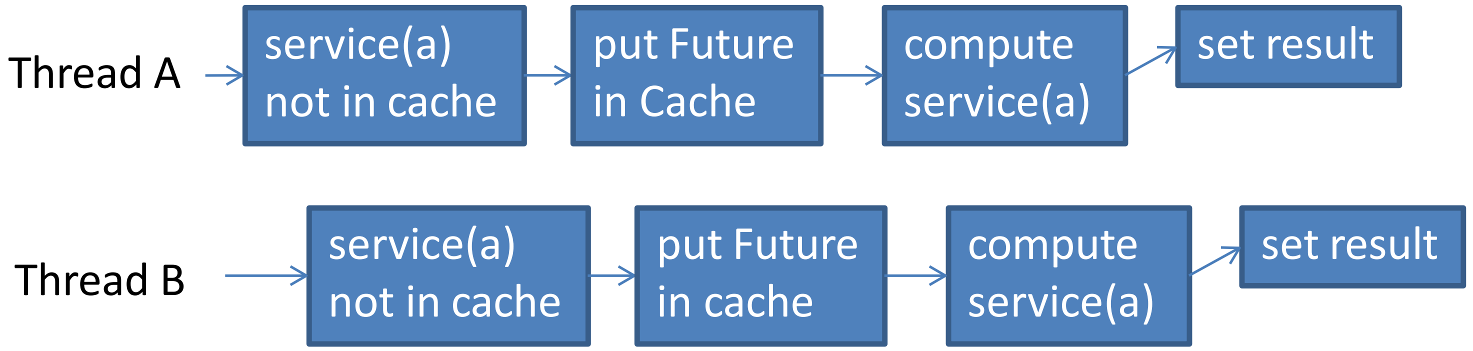


This is a big problem if factor(a) is required to execute once-and-only-once.

Click here for a sample program: CacheV2.java

Example

- Minor problem



This is a big problem if factor(a) is required to execute once-and-only-once.

Cohort Exercise 2

- Fix the minor problem on the previous slide. Notice that you can’t apply client-side locking on ConcurrentMap.
  - How do we make sure that factor(a) is executed once only for any a?

Tasks

- Most concurrent applications are organized around the execution of tasks: abstract, discrete units of work.
- Designing your program organization around tasks
  - Identify sensible task boundaries
  - Ideally, tasks are independent activities: work that does not depend on other tasks
  - Independence facilitates concurrency

Example: Server

- Most server applications offer a natural choice of task boundary: individual client requests.
- Executing tasks sequentially

```
public class SingleThreadWebServer {
        public static void main (String[] args) throws Exception {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
                Socket connection = socket.accept();
                handleRequest(connection);
        }
}
```

- This might work if *handleRequest* returns immediately – not for real world web server.

One Thread Per Task

- A more responsive approach is to create a new thread for servicing each request

```
class ThreadPerTaskWebServer {
    public static void main (String[] args) throws Exception {
    ServerSocket socket = new ServerSocket(80);
    while (true) {
        final Socket connection = socket.accept();
        Runnable task = new Runnable () {
            public void run() {
                handleRequest(connection);
            }
        };
        new Thread(task).start();
    }
}
```

One Thread Per Task

- Task processing is offloaded from the main thread – more responsive.
- Tasks can be processed in parallel – improved throughput.
- Task-handling code must be thread-safe, because it may be invoked concurrently for multiple tasks.

It works under light or moderate load.
Example: ThreadPerTaskWebServer.java

Unbound Thread Creation

- For production purposes (large webservers for instance) task-per-thread has some drawbacks.
  - Thread creation and tear down involves the JVM and OS. For lots of lightweight threads this is not very efficient.
  - Active Threads consume extra memory, for instance to provide for a thread stack.
  - If there are less CPU's than threads, some threads sit idle, consuming memory.
  - There is a limit on how many threads you can have concurrently. If you hit this limit your program will most likely become unstable.

The Executor Framework

- Single thread
  - poor responsiveness and throughput
- Thread-per-task
  - Poor resource management (consider a deny of service attack)
- The executor framework offers flexible thread pool management

Executor

```
public interface Executor {
    void execute (Runnable command);
}
```

- Executor provides a standard means of decoupling task submission from task execution.
  - The Runnable is the task itself.
  - The method execute defines how it is executed.

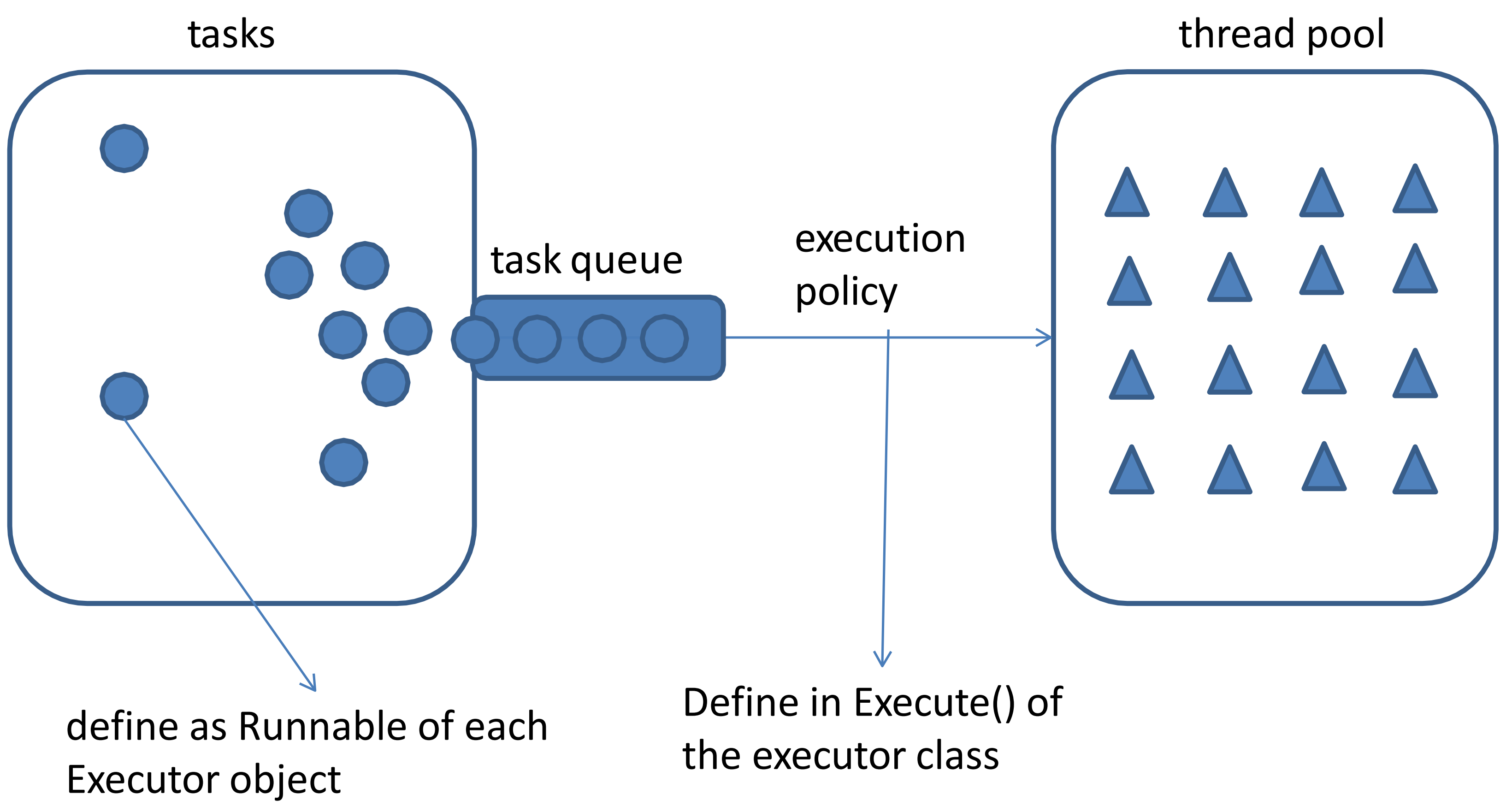
Example: ExecutorWebServer.java

Execution Policy

- Decoupling submission from execution is that it lets you specify the execution policy for a given class of tasks.
  - In what thread will tasks be executed?
  - In what order should tasks be executed (FIFO)?
  - How many tasks may execute concurrently?
  - How many tasks may be queued pending execution?
  - If a task has to be rejected because the system is overloaded, which task should be selected and how the application be notified?
  - What actions should be taken before or after executing a task?

Example:
SequentialExecutorWebServer.java
ThreadPerTaskExecutorWebServer.java

Thread Pools



Advantage of Thread Pools

- Reusing an existing thread; reduce thread creation and teardown costs.
- No latency associated with thread creation; improves responsiveness.

By properly tuning the size of the thread pool, you can have enough threads to keep the processors busy while not having so many that your application runs out of memory or thrashes due to competition among threads for resources

Thread Pool Implementations

- newFixedThreadPool
  - Fixed-size thread pool; creates threads as tasks are submitted, up to the maximum pool size and then attempts to keep the pool size constant
- newCachedThreadPool
  - Boundless, but the pool shrinks and grows when demand dictates so
- newSingleThreadExecutor
  - A single worker thread to process tasks, sequentially according to the order imposed by the task queue
- newScheduledThreadPool
  - A fixed-size thread pool that supports delayed and periodic task execution.

Example: ScheduledThreadPool.java

Cohort Exercise 4

- Modify ThreadPerTaskExecutorWebServer.java to use newFixedThreadPool with 100 threads. Name your program ExecutorWebServer.java.
- Compare its performance with the sequential and thread-per-task web server using MultipleClient.java (with the smaller number) with 10, 100, 1000 client threads.

Executor Lifecycle

- Shut down an Executor through ExecutorService

```
public interface ExecutorService extends Executor {
        void shutdown();
        List<Runnable> shutdownNow();
        boolean isShutdown();
        boolean isTerminated();
        boolean awaitTermination(long timeout, TimeUnit unit)
                throws InterruptedException;
        // … additional convenience methods for task submission
}
```

shutdown() vs shutdownNow()

- shutdown()
  - will just tell the executor service that it can't accept new tasks, but the already submitted tasks continue to run
- shutdownNow()
  - will do the same AND will try to cancel the already submitted tasks by interrupting the relevant threads. Note that if your tasks ignore the interruption, shutdownNow() will behave exactly the same way as shutdown().

Task Coupling and Execution Policy

- Thread pools work best when tasks are homogeneous and independent.
  - Dependency between tasks in the pool creates constraints on the execution policy which might result in problems (deadlock, liveness hazard, etc.)
  - Long-running tasks may impair the responsiveness of the service managed by the Executor.
  - Reusing threads create channels for communication between tasks – don’t use them.

Sizing Thread Pools

- The ideal size for a thread pool depends on the types of tasks and the deployment system
  - If it is too big, performance suffers
  - If it is too small, throughput suffers
- Heuristics
  - For compute intensive tasks, N+1 threads for a N-processor system
  - For tasks including I/O or other blocking operations, you want a larger pool

Optimal CPU Utilization

Given these definitions:

N = number of CPUs

U = target CPU utilization

W/C = ratio of wait time to compute time

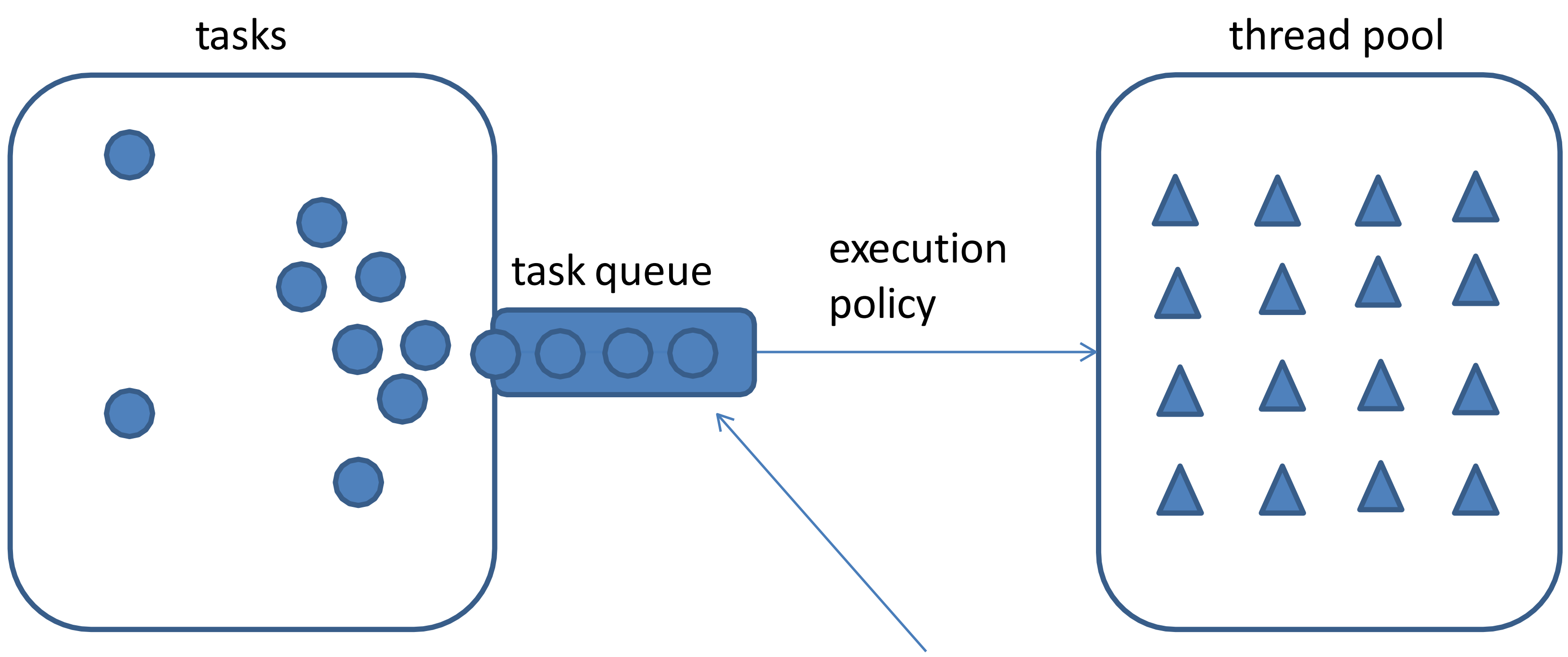
The optimal pool size is:

M = N * U * (1 + W/C)

The number of CPUs can be obtained by:

Runtime.getRuntime().availableProcessors()

Sizing the Queue



The queue may grow unbounded!
When a bounded task queue fills up, the saturation policy comes into play.
The default: throw RejectedExecutionException

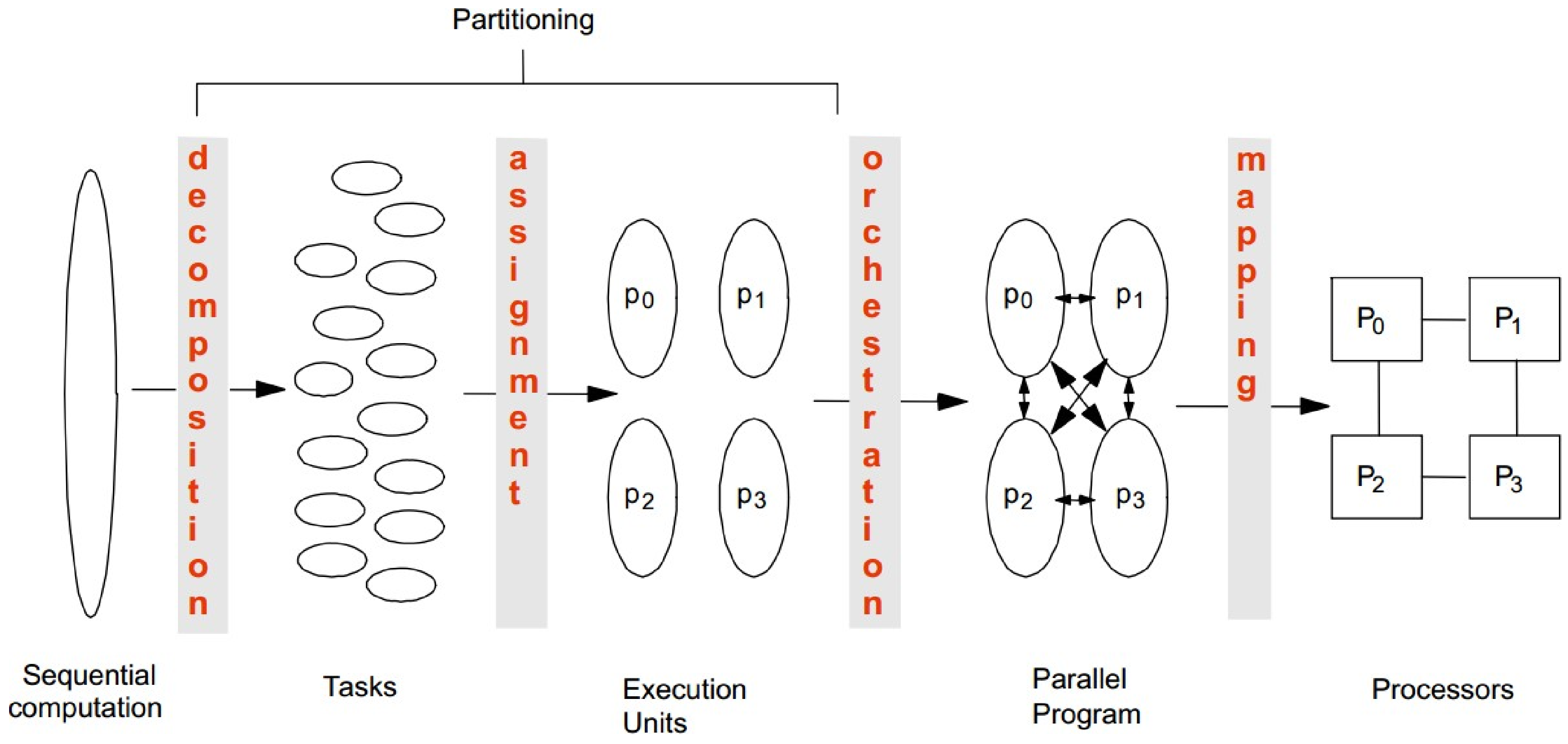
More Than CPUs

- Other resources that can contribute to sizing constraints are memory, file handles, socket handles, database connections, etc.
  - Add up how much of those resources each task requires and divide that into the total quantity available
- Alternatively, the size of the thread pool can be tuned by running the application using different pool sizes and observing the level of CPU and other resource utilization.

Finding Exploitable Parallelism

- The executor framework makes it easy to submit and execution tasks as well as specify an execution policy.
- How do you define the tasks such that you can get the maximum performance?

Common Steps to Parallelization



Decomposition

- Break up the computation into "self-contained" tasks to be divided among processes
  - Tasks shouldn't be too small or too big
    - Too small: the ratio of useful work vs overhead becomes small
    - Too big: Number of tasks available at a time is upper bound on achievable speedup
  - Tasks may become available dynamically

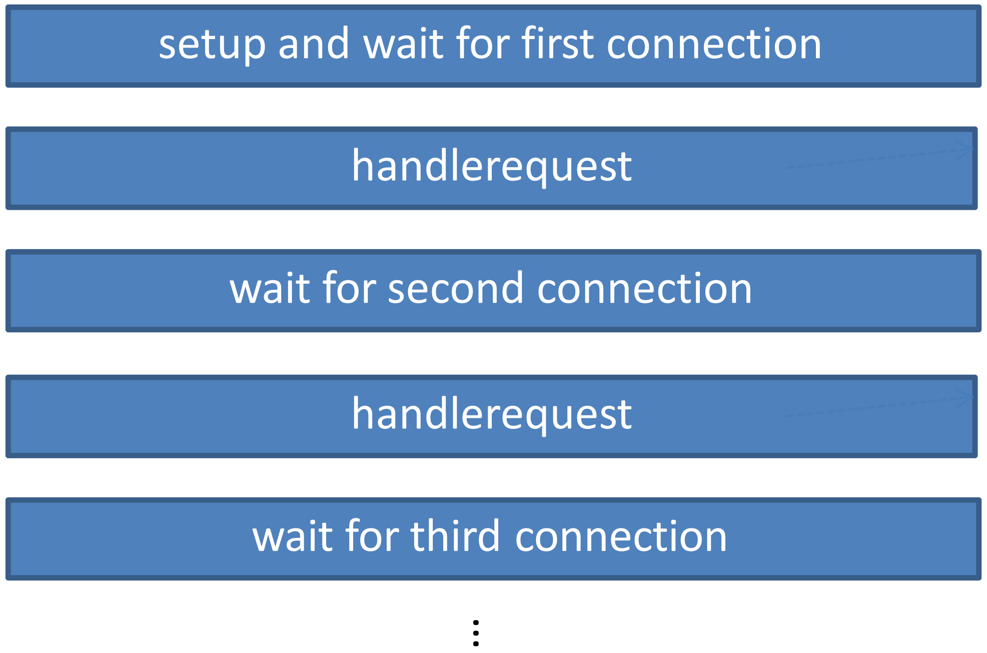
Assignment

- Specify mechanism to divide work among cores
  - We may want to balance the amount of work for each core
  - and reduce communication between the threads
- Structured approaches usually work well
  - Code inspection or understanding of application
  - Apply well-known design patterns

Orchestration and Mapping

- Figure out what kind of communication is needed between each pair of threads
  - Less communication is better: preserve locality of data
- Schedule the threads to satisfy tasks dependences
- Use Executor

Example: FactorWebserver


setup and wait for first connection
handlerequest
wait for second connection
handlerequest
wait for third connection
⋮

T1 and T2 dont have to wait for each other. Independent.
W1 n W2 = W1 n R2 = R1 n M2 = null

W1 & W2 -> no shared variable
W1 & R2 -> no shared | R1 & W2 -> no shared

Dependency

Bernstein's Condition

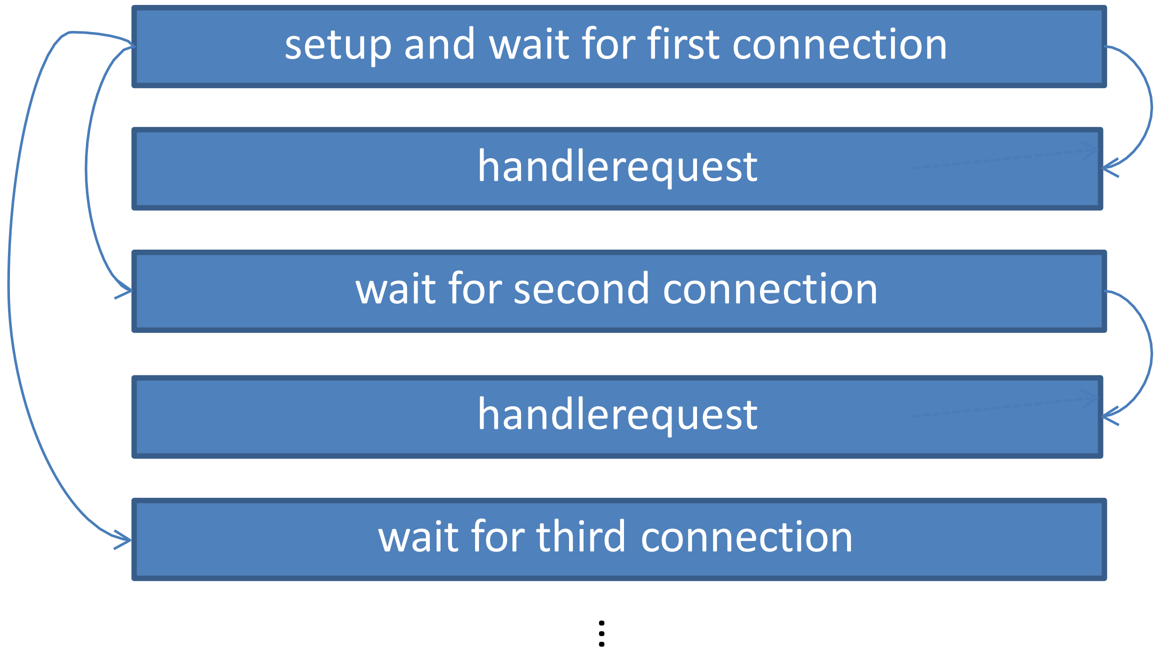
- Ri: set of memory locations read (input) by task Ti
- Wj: set of memory locations written (output) by task Tj

Two tasks T1 and T2 can run in parallel if input to T1 is not part of output from T2; and input to T2 is not part of output from T1 outputs from T1 and T2 do not overlap.

Example: If T1: a = x+y and T2: b = x+z, then R1 = {x, y}; W1 = {a}; R2 = {x, z}; W2 = {b}

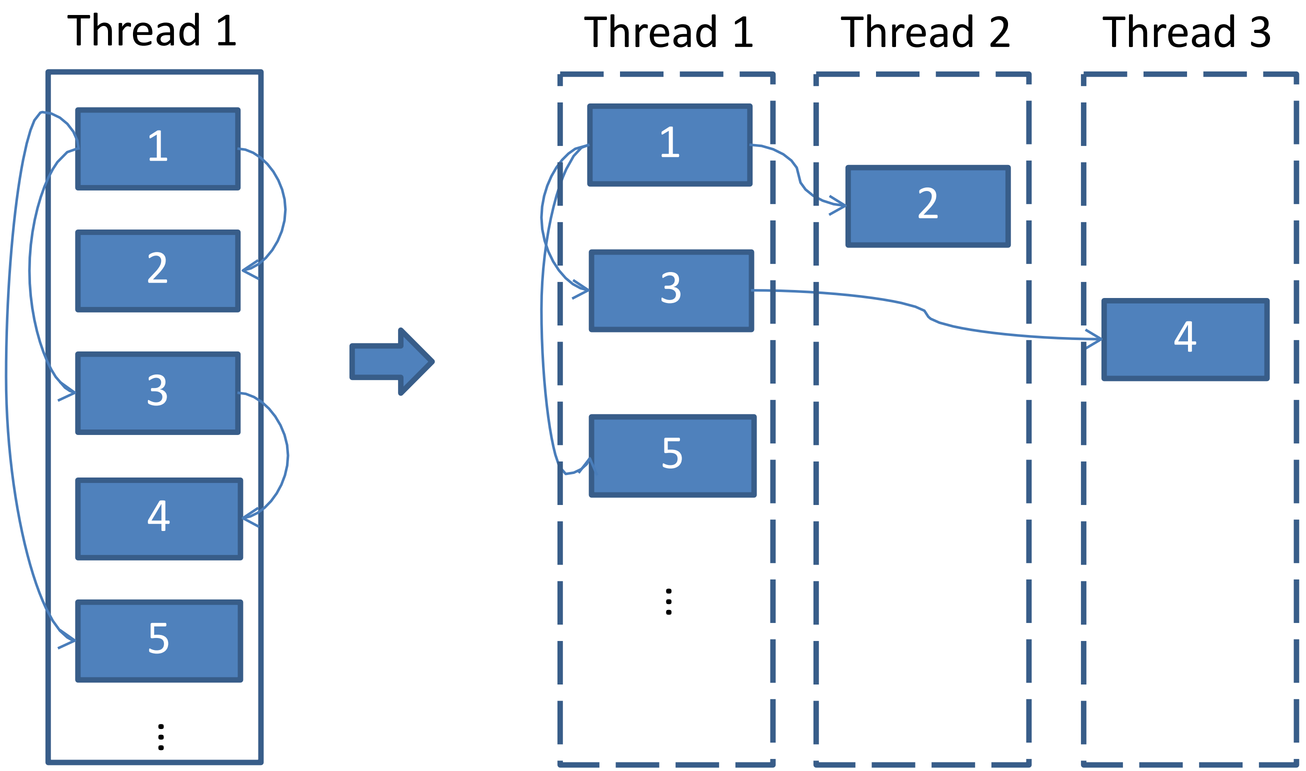
*** assume that x, y and z are different memory locations

Example: FactorWebserver


setup and wait for first connection
handlerequest
wait for second connection
handlerequest
wait for third connection
⋮

Example: FactorWebserver

- calls of *handlerequest* can run in parallel.



Orchestration Revisited

- Given a collection of concurrent tasks, what are the important considerations in mapping tasks to units of execution (e.g., threads)?
  - Magnitude of number of execution units platform will support (less a problem if you use Executor)

What if we can only have maximum 4 threads for the server?

  - Cost of sharing information among execution units

Consider parallelizing the Depth-First Search algorithm

Patterns for Parallelization

- Provides a cookbook to systematically guide programmers
  - Decompose, Assign, Orchestrate, Map
  - Can lead to high quality solutions in some domains
- Provide common vocabulary to the programming community
  - Each pattern has a name, providing a vocabulary for discussing solutions
- Helps with software reusability, malleability, and modularity
  - Written in prescribed format to allow the reader to quickly understand the solution and its context

Patterns for Parallelization

- "Patterns for Parallel Programming", Mattson, Sanders, and Massingill (2005).
- The patterns could help you in
  - Exposing concurrent tasks
  - Mapping tasks to processes to exploit parallel architecture
  - Providing supporting code and data structures
  - Providing low-level mechanisms needed to write parallel programs

Single Program, Multiple Data

- All threads/processes run the same program, operating on different data. This model is particularly appropriate for problems with a regular, predictable communication pattern.
  - MATLAB supports SPMD blocks.
  - Often adopted for GPU programming

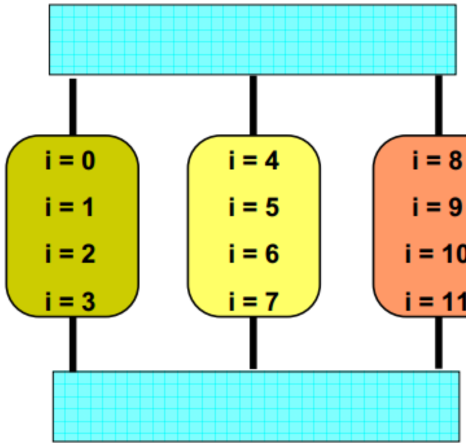
Cohort Exercise 6

- Given Exercise1.java, apply SPMD (Single Program, Multiple Data) design pattern for concurrent programming to parallelize the program which approximates π by integration.

Loop Parallelism Pattern

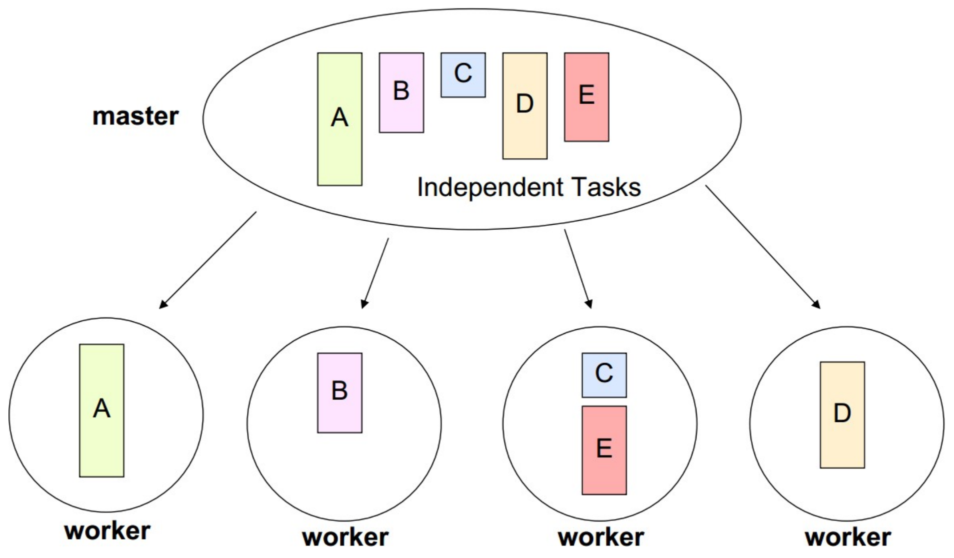
- Given a loop, each thread/process execute part of the loop.
  - Programming models like OpenMP provide directives to automatically assign loop iteration to execution units

```
#pragma omp parallel for
for(i = 0; i < 12; i++)
       C[i] = A[i] + B[i];
```



Master/Worker Pattern

- A master thread/process divides a problem into several sub-problems and dispatches them to several worker processes.

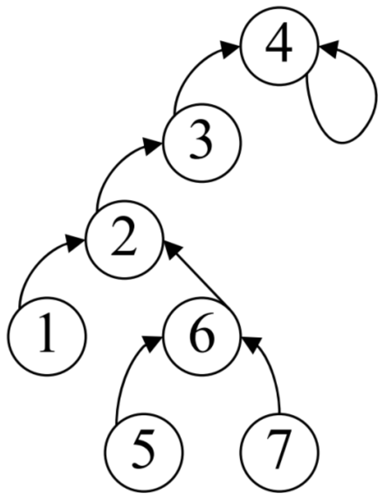


Fork/Join Pattern

- Tasks are created dynamically
- Tasks can create more tasks
  - Manages tasks according to their relationship
  - Parent task creates new tasks (fork) then waits until they complete (join) before continuing on with the computation

Example: PRAM

- How to parallelize BFS?
  - Start from a root, and visit all the connected nodes in a graph
  - Nodes closer to the root are visited first
  - Nodes of the same hop-distance (level) from the root can be visited in parallel



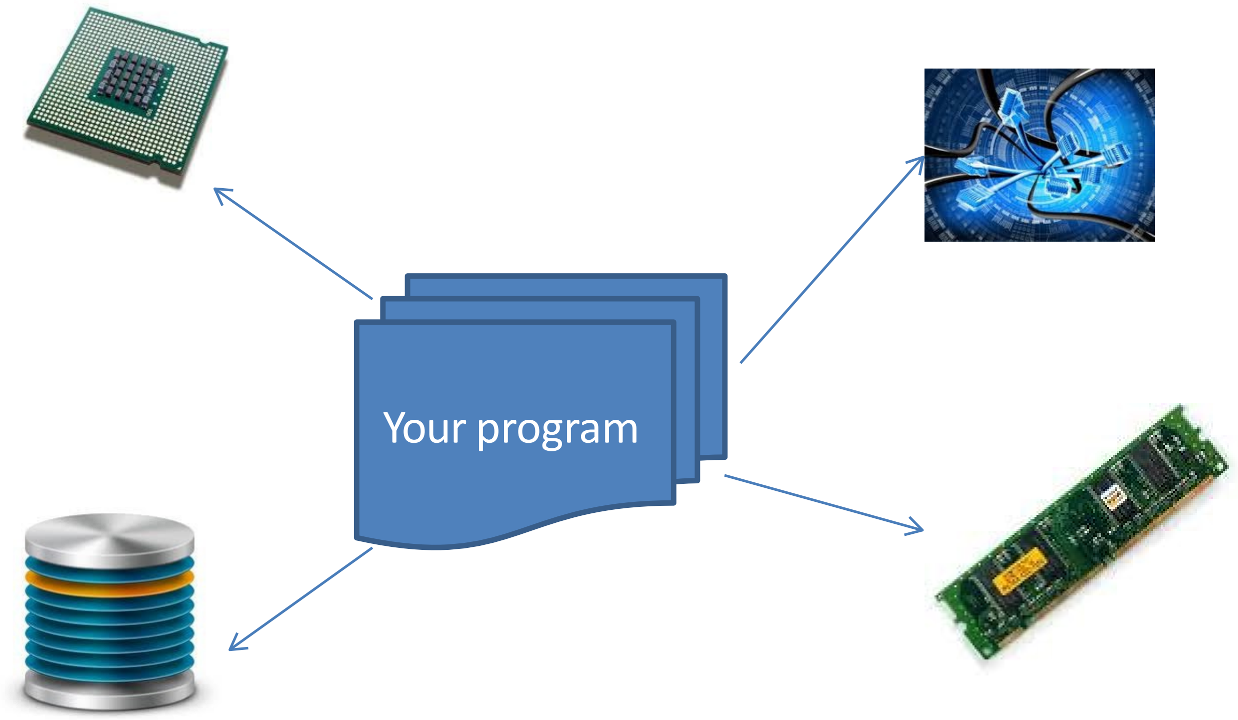
Performance vs Complexity

- One of the primary reasons to use threads is to improve performance.
- Techniques for improving performance also increase complexity and the likelihood of safety and liveness failures.

Avoid premature optimization. First make it right, then make it fast – if it is not already fast enough.

The quest for performance is probably the single greatest source of concurrency bugs.

Bottleneck



Find out what the bottleneck is before you optimize.

Amdahl’s Law

- Amdahl’s law says that on a machine with N processors, we can achieve a speed up of at most:

$$speedup \leq \frac{1}{F + \frac{1-F}{N}}$$

where F is the fraction of the calculation that must be executed serially.

Serialization

```
class Slave extends Thread {
   private final BlockingQueue<Runnable> queue;
   private final ExecutorService exec = Executors.newFixedThreadPool(100);


   public Slave (BlockingQueue<Runnable> queue) {
          this.queue = queue;
   }

   public void run() {
          while (true) {
               try {
                      exec.execute(queue.poll());
               }
               catch (Exception e) {
                      break;
               }
          }
   }
}
```

All Concurrent applications have some sources of serialization

Cost Introduced by Threads

- Context switching: requires saving the execution context of the currently running thread and restoring the execution context of the newly scheduled thread
  - CPU time spent on JVM/OS
  - Cache misses
  - Costs about 5,000 to 10,000 clock cycles
- Memory synchronization:
  - Memory barriers inhibit compiler optimization

Lock/Release Cost

- Access to resources guarded by an exclusive lock is serialized – one thread at a time delay may access it

```
public String getNames() {
        List<String> names = new Vector<String>();
        names.add(“Alice”);
        names.add(“Bob”);
        names.add(“Carl”);
        return names.toString();
}
```

- Example: a naïve execution would require and release the lock on the vector four times.
- It gets much worse with lock contention.

Lock Contention

- Two factors influence the likelihood of contention for a lock
  - How often that lock is requested
  - How long it is held once acquired
- There are three ways to reduce lock contention
  - Reduce the duration for which locks are held
  - Reduce the frequency with which locks are requested
  - Replace exclusive locks with coordination mechanisms that permit greater concurrency

“Get in, get out”

```
public class ReduceLockScope {
    //@GuardedBy("this")
    private final Map<String, String> attributes = new HashMap<String, String>();

    public synchronized boolean userLocationMatches (String name, String regexp) {
        String key = "users." + name + ".location";

        String location = attributes.get(key);

        if (location == null) {

            return false;
        }
        else {
            return Pattern.matches(regexp, location);
        }
    }
}
```

“Get in, get out”

```
public class ReduceLockScope {
    //@GuardedBy("this")
    private final Map<String, String> attributes = new HashMap<String, String>();

    public boolean userLocationMatches (String name, String regexp) {
        String key = "users." + name + ".location";
        String location;
        synchronized (this) {
            location = attributes.get(key);
        }
        if (location == null) {
            return false;
        }
        else {
            return Pattern.matches(regexp, location);
        }
    }
}
```

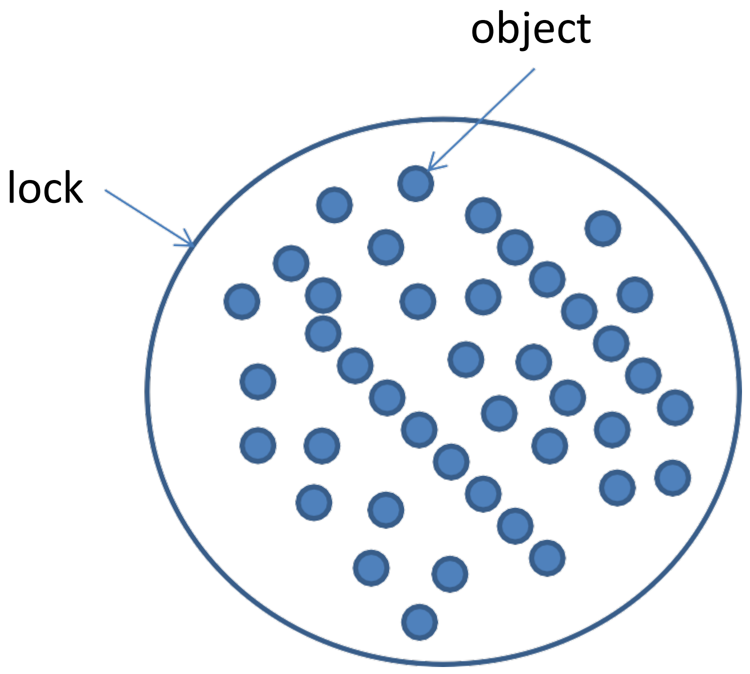
The amount of serialized code is reduced!

“Get in, get out”

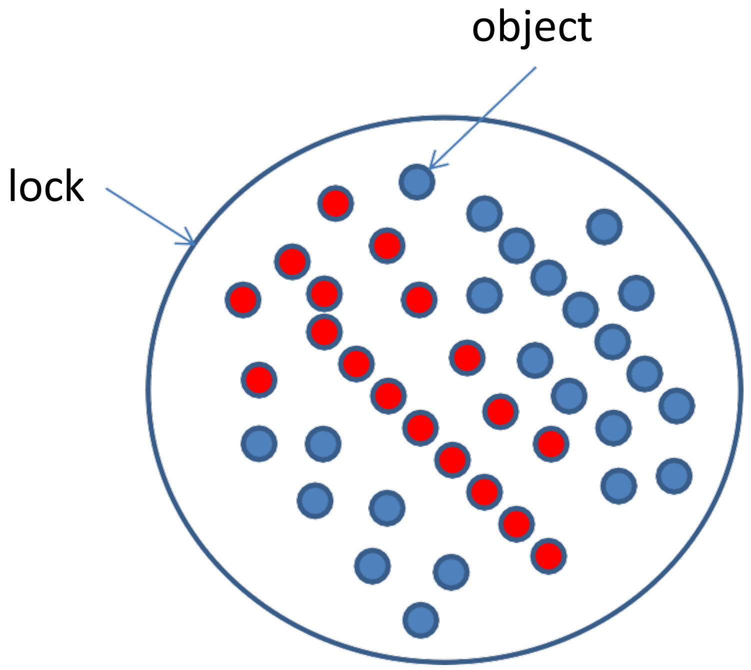
```
public class ReduceLockScope {
    //@GuardedBy("this")
    private final Map<String, String> attributes = new ConcurrentHashMap<String,  String>();

    public boolean userLocationMatches (String name, String regexp) {
            String key = "users." + name + ".location";
            String location = attributes.get(key);

            if (location == null) {
                    return false;
            }
            else {
                   return Pattern.matches(regexp, location);
            }
    }
}
```

Delegating thread safety to ConcurrentHashMap

Reducing Lock Granularity

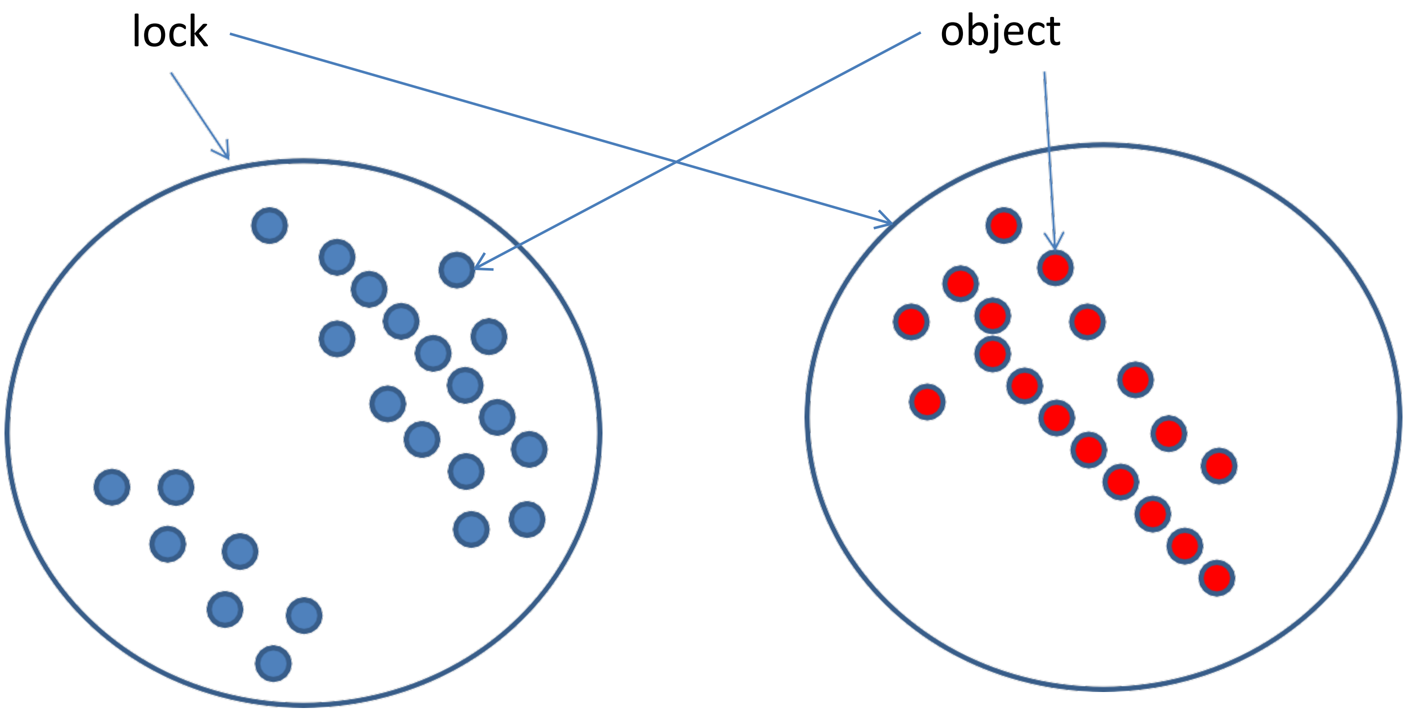


Every thread acquires the lock to access any locked object



Every thread acquires the lock to access any locked object

Reducing Lock Granularity



Lock splitting

Lock Splitting

```
public class ServerStatus {
	public final Set<String> users; //@GuardedBy("this")
	public final Set<String> queries; //@GuardedBy("this")
	...
	public synchronized void addUser(String u) {
		users.add(u);
	}
	public synchronized void addQuery(String q) {
		queries.add(q);
	}
	public synchronized void removeUser(String u) {
		users.remove(u);
	}
}
```

Lock Splitting

```
public class ServerStatus {
        public final Set<String> users; //@GuardedBy(“users")
        public final Set<String> queries; //@GuardedBy(“queries")
        ...
        public void addUser(String u) {
                synchronzied (users) { users.add(u); }
        }
        public void addQuery(String q) {
                synchronized (queries) { queries.add(q); }
        }
        public synchronized void removeUser(String u) {
                synchronzied (users) { sers.remove(u); }
        }
}
```

Lock Stripping

- Lock splitting can sometimes to extended to partition lock on a variable sized set of independent object, which is called lock stripping.
  - Example: ConcurrentHashMap, 16 locks

Alternatives to Exclusive Locks

- To forego the use of exclusive locks in favor of a more concurrency-friendly means of managing shared state
  - Read-write locks: more than one reader can access the shared resource concurrently, but writers must acquire the lock exclusively
  - Immutable objects
  - Atomic variables

Cohort Exercise 8

Given StripedMap.java, assume that the objects in the buckets are independent. Complete method **get(), size()** and **clear()** using the idea of lock stripping.
// look at codes sheet

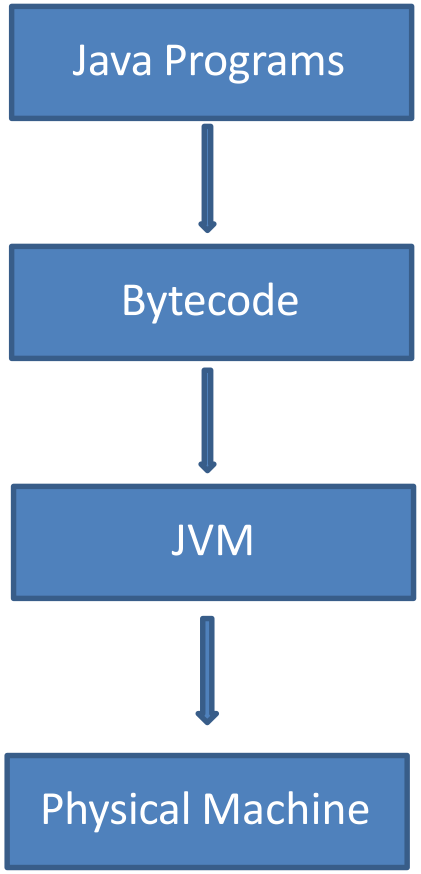
NON-BLOCKING SYNCHRONIZATION

Not tested

Disadvantage of Locking

- The ratio of scheduling overhead to useful work can be quite high when the lock is frequently contended – due to context switch and scheduling delays.
- A thread with the lock may be delayed (due to a page fault, scheduling delay, etc.).
- Locking is simply a heavyweight mechanism for simple operations like *count++*

Reality is Messy



What if we know what are the atomic steps?

What are the atomic steps? For example, how many steps are “i++”?

What are the order of execution? Given “i++; j++; i++”, can we switch the last two statements?

Where are the variable values stored? Cache, heap memory, stack memory.

Hardware Support for Concurrency

- Processors designed for multiprocessor operation provide special instructions for managing concurrent access to shared variables, for example:
  - compare-and-swap
  - load-linked/store-conditional
- OSs and JVMs use these instructions to implement locks and concurrent data structures

Compare and Swap

- CAS has three operands
  - a memory location V,
  - the expected old value A,
  - and the new value B.
- CAS updates V to the new value B, but only if the value in V matches the expected old value A; otherwise, it does nothing. In either case, it returns the value currently in V.

Click here for a sample program: SimulatedCAS.java

A Non-blocking Counter

```
public class CasCounter {
    private SimulatedCAS value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        } while (v != value.compareAndSwap(v, v + 1));
        return v + 1;
    }
}
```

Is it thread-safe?
Is it more efficient than a lock-based counter?
Any potential problem?

CAS is used to implement AtomicXxx in Java

CAS in Java

- CAS is supported in atomic variable classes (12 in java.util.concurrent.atomic), which are used, to implement most of the classes in java.util.concurrent
  - AtomicInteger, AtomicBoolean, AtomicReference, etc.

Non-blocking Algorithms

- An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread;
- Non-blocking algorithms are immune to deadlock (though, in unlikely scenarios, may exhibit livelock or starvation)
- Non-blocking algorithms are known for
  - Stacks (Treiber’s), queues, hash tables, etc.