Parallel Program Optimization

Shuhao Zhang

Nanyang Technological University

shuhao.zhang@ntu.edu.sg

June 4, 2024

Parallel Program Optimization

- The optimization of parallel programs can be tedious.
  - Different optimization targets: *total execution time*, *energy-efficiency*, *resource utilization*, *minimize latency violation*, *better scalability*, etc.
  - Different optimization strategies lead to tradeoffs among: *more parallelism* vs. *more chances of bugs*, e.g., deadlocks; *closer to optimization target* vs. *programming/code maintenance complexity*;

The Optimization of Parallel Programs

We discussed various mechanism to:

1. **["Open source": Improves Concurrency]** handling *lock contentions* to gain higher concurrency
2. **["Reduce expenditure": Improves Resource Utilization]** improve *resource utilization* to reduce computing resource wastage

Figure: Open source and reduce expenditure

Lock is bad

- 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++.

Lock is bad: 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
  - Apart from using locks, we can also use violate keyword to achieve memory synchronization in Java.
  - FYI: In C/C++, we can use "memory_fence" statement.
- Waiting time:
  - a thread maybe blocked from execution due to locks leading to waiting time (or synchronization time)

Lock is bad: Lock/Release Cost

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

Example

A naive execution would require and release the lock on the vector four times in the following example. It can get much worse with lock contention (i.e., multiple threads access to the vector).

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

Question

How to revise it?

Lock Contention

- Two factors influence the likelihood of contention for a lock
  - How *long* it is held once acquired
  - How *often* that lock is requested

Dealing with Lock Contention Overview

There are three common ways to deal with lock contention

1. Reduce the duration for which locks are held
2. Reducing lock granularity
3. Replace exclusive locks with coordination mechanisms that permit greater concurrency

Reduce Lock Duration

Reduce the duration for which locks are held is a common approach to improve efficiency of parallel program.

Example: User Location Matches

- Assuming a simple program to determine if a user is at a location.
- Suppose we have implemented a userLocationMatches method...

Narrowing Lock Scope

```
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);
        }
    }
}
```

Question

Which part(s) are in critical section? See ReduceLockScope.java.

Narrowing Lock Scope

- The entire process of execution of the method holds a lock
- The only operation that really needs to ensure synchronization is the
  - String location = attributes.get (key);
- So most of the holding time is wasted.

Narrowing Lock Scope Revised

```
public class ReduceLockScope_revised {
    //@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);
        }
    }
}
```

Question

Why it helps? See ReduceLockScope_revised.java. We will come back to this in Tutorial.

Summary of Reducing Lock Duration

- While shrinking synchronized blocks can improve scalability, a synchronized block can't be too small
  - operations that need to be atomic (such updating multiple variables that participate in an invariant) must be contained in a single synchronized block.
  - recall the "compound action" examples we have seen earlier?
- And, because the cost of synchronization is non-zero, breaking one synchronized block into multiple synchronized blocks (correctness permitting) at some point becomes counterproductive in terms of performance.
- The ideal balance is of course platform-dependent, but in practice it makes sense to worry about the size of a synchronized block only when you can move "substantial" computation or blocking operations out of it.

Reduce Lock Granularity

- Another way to reduce the holding time of locks is to reduce the frequency with which threads request locks (thus reducing the possibility of contention).
- This can be achieved by techniques such as *lock decomposition/splitting* and *lock segmentation*, in which a plurality of mutually independent locks are used to protect independent state variables, thus changing the situation that these variables were previously protected by a single lock.
- These technologies can reduce the granularity of lock operation and achieve higher scalability. However, the more locks are used, the higher the risk of **deadlock**.
  - luckily though, you are now masters to deal with deadlock ;)

Motivating Example: Synchronized Classes

- Why is this not ideal?

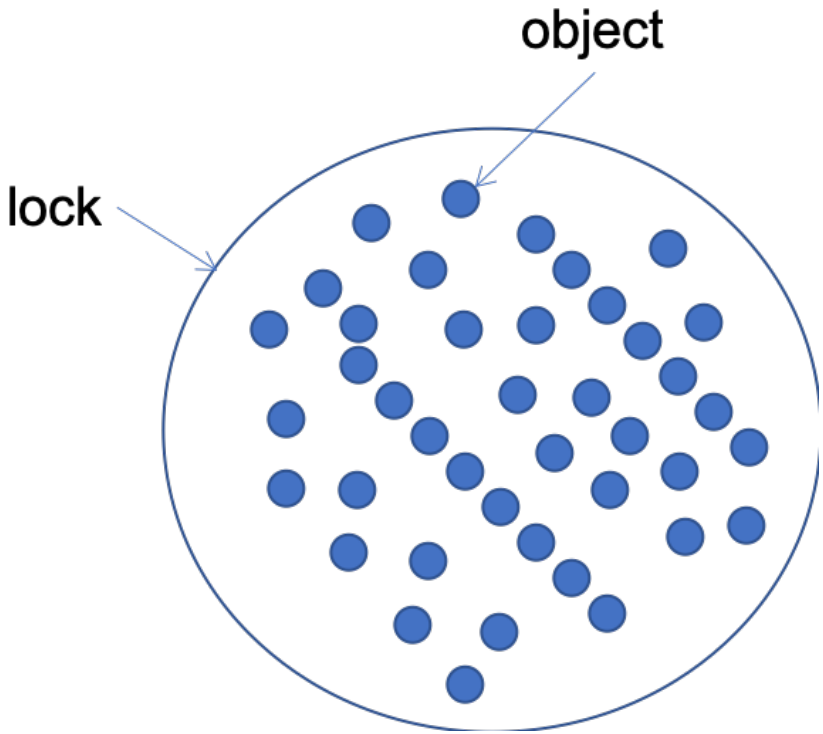
```
public static void doSomething (Vector list) {
        synchronized (list) {
                for (int i = 0; i < list.size(); i++) {
                    doSomething(list.get(i));
                }
        }
}
```

Synchronized Classes

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

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

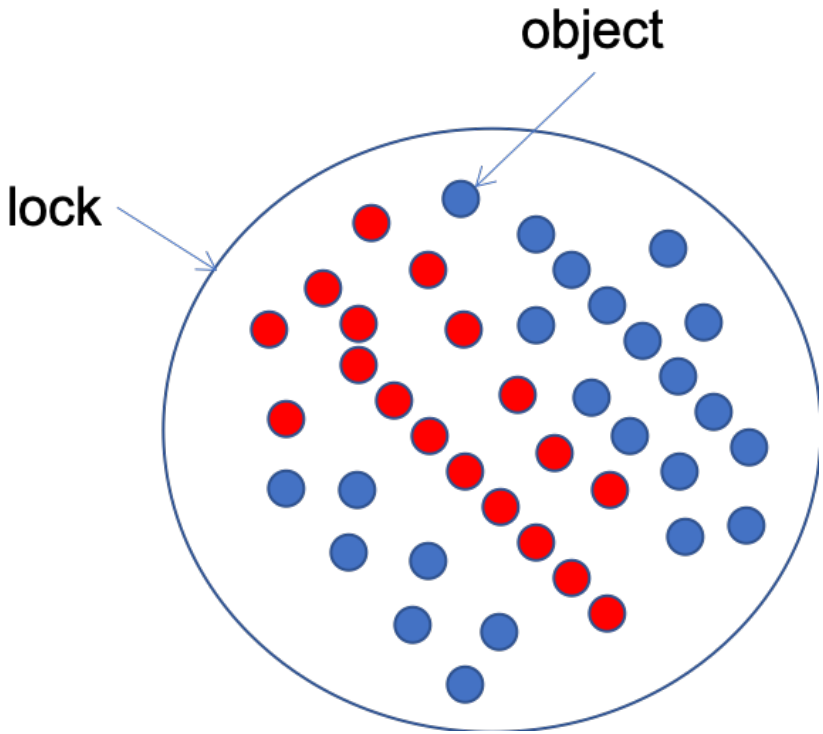
Lock Decomposition



Example

Every thread acquires the lock to access any locked object

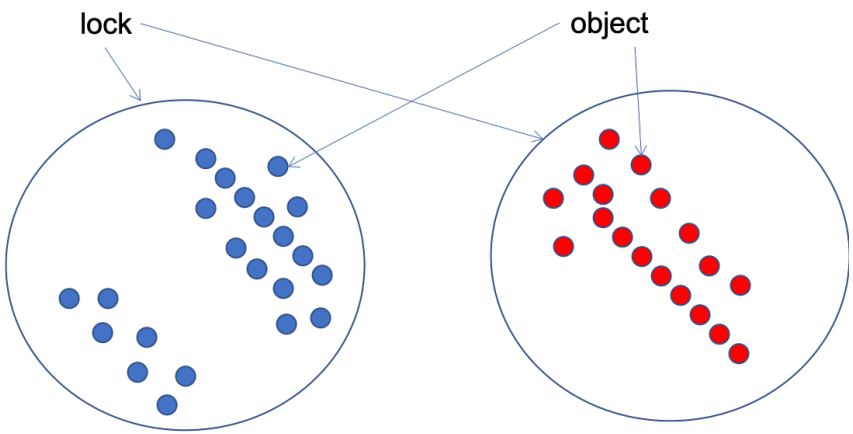
Lock Decomposition



Example

Every thread acquires the lock to access any locked object

Lock Decomposition



Lock splitting

Example of Lock Decomposition

```
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);
        }
}
```

Example of Lock Decomposition

```
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); }
        }
}
```

Notes

addUser() only touches lock of ‘users’, and will not interfere with addQuery().

More than Lock Decomposition: Lock Segmentation/Stripping

- Lock decomposition can sometimes to extended to partition lock on a variable sized set of independent object, which is called lock segmentation/stripping.

Example of Lock Stripping: 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**.

Notes

See more in SimpleConcurrentHashMap.java.

Can we remove locks?

- Lock is Bad, and we revise it by: reduce duration; reduce granularity.
- Can we remove it at all? – Replace exclusive locks with coordination mechanisms that permit greater concurrency

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

Copy on Write

- 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.

Example: 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.
- See the CopyOnWriteArrayListExample.java.

Note

It is also widely recognized as "immutable operation mechanism".

Revise the algorithm to get rid of locks

- 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.

Example 1: Live with Race Condition w/o Lock

- Race condition can be benign.
  - Sometimes, it is even intended in order to get better performance.
- Revisit the FactorTread.java, see the ‘found‘ variable, what do you see?
  - It is a global variable
  - It may be read/write by multiple threads concurrently
  - There is no ‘violate‘ nor locks to protect the access to it

Question

Can you point out why it is fine to live with such race condition in this case?

Example 2: Hardware Support for Higher 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.

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 package
- AtomicInteger, AtomicBoolean, AtomicReference, etc.
- See CompareAndSwapLock.java.

Native Implementation

```
public final native
@MethodHandle.PolymorphicSignature
@HotSpotIntrinsicCandidate
boolean compareAndSet(Object... args);
```

Simulated CAS

```
public class SimulatedCAS {
    private int value;
    public synchronized int get() {
        return value;
    }
    public synchronized int compareAndSwap(int expectedValue,
                                           int newValue) {
        int oldValue = value;
        if (oldValue == expectedValue)
            value = newValue;
        return oldValue;
    }
    public synchronized boolean compareAndSet(int expectedValue,
                                              int newValue) {
        return (expectedValue
                == compareAndSwap(expectedValue, newValue));
    }
}
```

The above is just a simulation of CAS. CAS is usually provided as a single instruction by modern processors (and JVM).

Example of using CAS: 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;
    }
}
```

Parallel Program Optimization

- The optimization of parallel programs can be tedious.
- We have discussed various mechanisms to deal with **lock contentions** – a killer reason for performance issues of parallel program.
- In the following, we discuss several mechanisms to further improve **resource utilization**:
  1. *Avoid busy waiting*
  2. *Caching*
  3. *Future task*

The Pitfalls of Busy Waiting

- Busy waiting is not efficient
  - Consider a voting system with two threads. One collects votes and the other is waiting to count the votes when the voting is completed
  - Example program: Voting.java

Example: Voting System

```
public class Voting {
        public static void main(String args[]){
                VoteCounter counter = new VoteCounter ();
                counter.start();
                VoteCollecter collector = new VoteCollecter (counter);
                collector.start();

                try {
                        collector.join();
                        counter.join();
                } catch (InterruptedException e) {
                        System.out.println("some thread is not finished"
                            );
                }
        }
}
```

VoteCollecter

```
class VoteCollecter extends Thread {
        private VoteCounter counter;

        public VoteCollecter (VoteCounter counter) {
                this.counter = counter;
        }

        public void run () {
                System.out.println ("Voting and collecting.");
                int[] votes = new int[10000];

                for (int i = 0; i < 10000; i++) {
                        votes[i] = (int) (100*Math.random());
                }

                counter.setVotes(votes);
                System.out.println ("Voting finished. Cleaning up.");
        }
}
```

VotingCounter - Option 1

```
while (true) {
    synchronized (this) {
        if (votes != null) {
            break;
        }
    }
}
```

is this a good design?

- This code repeatedly enters a synchronized block to check if votes is not null.
- It wastes CPU cycles because it continuously locks and unlocks the object.
- This can cause high CPU usage, known as a “busy wait”.
- Not an efficient way to wait for a condition to be met.

VotingCounter - Option 2

```
synchronized (this) {
        while (true) {
                if (votes != null) {
                        break;
                }
        }
}
```

how about this?

- This code enters a synchronized block and then performs a busy wait within it. The while (true) loop runs indefinitely until votes is not null.
- The 'wait()' method is never called, so this will also lead to high CPU usage.
- Additionally, since it holds the lock while busy-waiting, it prevents any other thread from accessing synchronized methods or blocks on this object, which can lead to deadlock or other threads being blocked.

Wait and Notify

- Busy waiting is not efficient
- Use wait()/nofityAll() to avoid busy waiting.
- The C++ equivalent is *condition_variable*.

VotingCounter - Option 3

```
synchronized (this) { //mind the order of synchronized and while
        while (votes == null) { //Q: why a while loop is needed here?
                try {
                        wait();
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
```

how about this?

- This is the correct way to wait for a condition to be met. The while (votes == null) loop checks the condition and calls wait() if votes is still null.
- The wait() method releases the lock and puts the thread into a waiting state until it is notified.
- When notifyAll() is called, the thread wakes up and checks the condition again.

VotingCounter - Option 4

```
while (votes == null) {
        synchronized (this) {
                try {
                        wait();
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
```

- This code attempts to wait for votes to be set, but it has a major flaw.
- The wait() call is placed inside a synchronized block, but this block is inside a loop that is outside any synchronization.
- Therefore, the condition votes == null is checked without synchronization, leading to a race condition.
- Multiple threads can enter the loop simultaneously, but only one thread can hold the lock and call wait() at a time. This can result in threads not properly waiting for the condition.
- It is essential to hold the lock while checking the condition and

To Share or Not to Share, That is the Question

- Multithreading is considered one of the most difficult topics in computer science, it requires a programmer not only to understand how to manage processes, but also how to manage **memory**.
  - Or more generally, manage **intermediate results**.
- So far, we shall be able to write multithreaded program to solve the same set of tasks (ideally) much faster than singlethread program by managing processes properly.
  - Results caching is another key technique we must be aware of in writing high-performance multithreaded program.

To Share or Not to Share, That is the Question

- Recall our early example of searching for prime factors.
- Suppose we have a server that continuously identifies factors of a given number submitted by multiple users.
  - Results can be reused if the input number has been resolved before (i.e., being cached).
  - So, once result is obtained, put it in a cache.

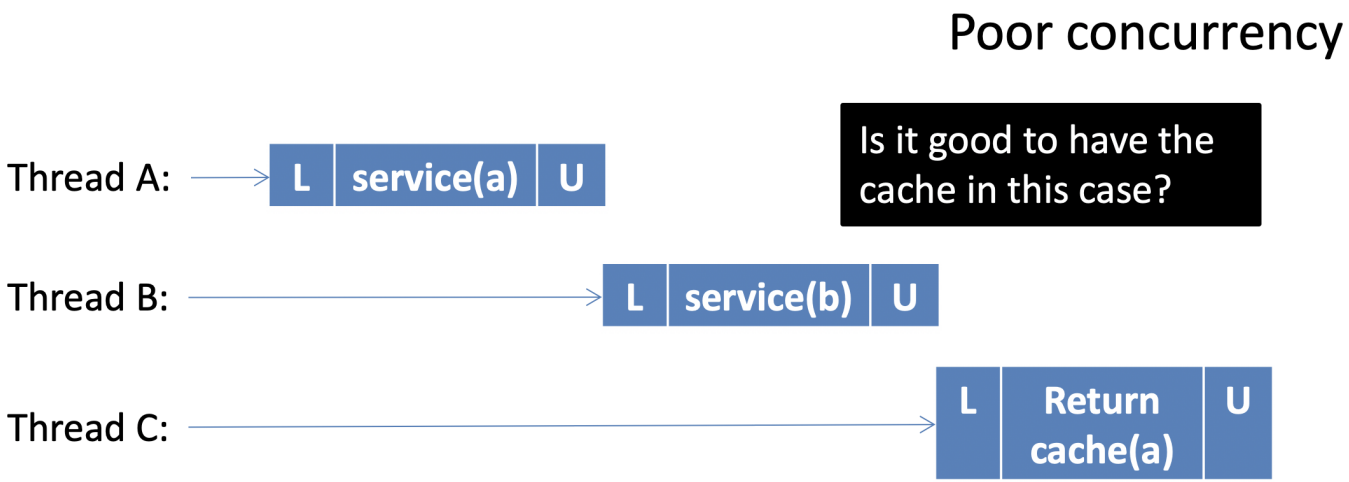
Prime factor searching server - v1

```
public class ResultCachingV1 {
    private final Map<Integer, List<Integer>> results = new HashMap<
        Integer, List<Integer>>();
    public synchronized List<Integer> service(int input) {
        List<Integer> factors = results.get(input);
        if (factors == null) {//haven't got the factors for the given
            input yet.
            factors = factor(input);//identify the factors.
            results.put(input, factors);//put the factors into results.
        }
        return factors;
    }
    public List<Integer> factor(int n) {
        List<Integer> factors = new ArrayList<Integer>();
        for (int i = 2; i <= n; i++) {
            while (n % i == 0) {factors.add(i); n /= i;}
        }
        return factors;
    }
}
```

Question

Is this design beneficial?

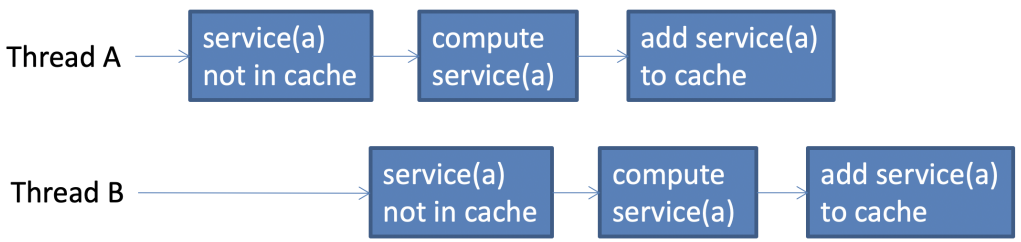
Prime factor searching server - v1



Question

Is this design beneficial? No...

Prime factor searching server - v2



Note

It still leads to potentially redundant computation.

Prime factor searching server

How do we make sure that factor(i) is executed once only for any input i, and can be used to serve future requests?

- Let's study a concept called "future task".

Future Task

- "extending Thread" and "implement Runnable", allows us to define multi-threaded task but the "run" method returns void.
  - What if we want to return value upon the task finish?
- Similar mechanism is supported in C++: <std::future>. https://www.cplusplus.com/reference/future/future/

Callable v.s. Runable

```
public class CallableTask implements Callable<String> {
    public String call() throws Exception {//run()
        Thread.sleep(10000);
        System.out.println("Executing call()!");
        return "success";
    }
}
```

- callabletask is the third way of defining multi-threaded tasks.
- Similar to the case when we define run(), but here, we can have return type.
- It is often used together with the future task mechanism.

FutureTask v.s. Thread

```
Thread a=new Thread(new
    Runnable() {...});
a.start();
```

```
FutureTask<String> future
    = new FutureTask<>(
    new CallableTask());
future.run();
```

Left: Thread and Runnable; **Right:** FutureTask and Callable;

FutureTask Complete Example

```
public class FutureTaskExample {
    public static void main(String[] args) {
        FutureTask<String> future = new FutureTask<>(new CallableTask())
            ;
        future.run();//thread.start();
        System.out.println("Result=");
        try {
            String result = future.get(1, TimeUnit.MILLISECONDS);//
                thread.join().
            System.out.println(result);
        } catch (InterruptedException | ExecutionException |
            TimeoutException e) {
            System.out.println("EXCEPTION!!!");
            e.printStackTrace();
        }
    }
}
```

FutureTask Complete Example

```
public class FutureTaskExample {
    public static void main(String[] args) {
        FutureTask<String> future = new FutureTask<>(new CallableTask())
            ;
        future.run();//thread.start();
        System.out.println("Result=");
        try {
            String result = future.get(1, TimeUnit.MILLISECONDS);//
                thread.join().
            System.out.println(result);
        } catch (InterruptedException | ExecutionException |
            TimeoutException e) {
            System.out.println("EXCEPTION!!!");
            e.printStackTrace();
        }
    }
}
```

- Future.get() returns the result immediately if 'the future is here' (Task is completed) and Blocks if the task is not complete yet.

Prime factor searching server revisited

- Let's now go back to the prime factor searching server implementation
- Let's properly utilize caching (with helps of future and callable).

Prime factor Searching Server - FutureTask Version

```
public class ResultCachingV3 {
        private final ConcurrentHashMap<Integer, Future<List<Integer>>>
            results = new ConcurrentHashMap<Integer, Future<List<
            Integer>>>();
        public List<Integer> service (final int input) throws Exception
            {
                Future<List<Integer>> f = results.get(input);
                if (f == null) {
                        Callable<List<Integer>> eval = new Callable<List
                            <Integer>>() {
                                public List<Integer> call () throws
                                    InterruptedException {
                                        return factor(input);
                                }
                        };
                        FutureTask<List<Integer>> ft = new FutureTask<
                            List<Integer>>(eval);
                        f = results.putIfAbsent(input, ft);
                        if (f == null) {
                                f = ft;
                                ft.run();
                        }
                }
                return f.get();
        }
        public List<Integer> factor(int n) {
        ...
        }
}
```