University of Cape Town

CSC2002S

Notes & Practised Questions 2022

Author: Bonga Njamela

Table of Contents

Section 1

Parallel and Concurrent Programming

A. Introduction to Parallelism and Concurrency

A.1 Parallelism vs Concurrency

- Parallel programs use extra computational resources to solve a problem faster (increasing throughput via simultaneous execution)
- Concurrent programs manage access to shared resources correctly and efficiently
- Parallel algorithm uses the FORALL loop which is like a *for* loop except that it does all the iterations in parallel

```
int sum(int[] arr){
  res = new int[4];
  len = arr.length;
  FORALL(i=0; i < 4; i++) { //parallel iterations
    res[i] = sumRange(arr,i*len/4,(i+1)*len/4);
  }
  return res[0]+res[1]+res[2]+res[3];
}
int sumRange(int[] arr, int lo, int hi) {
   result = 0;
   for(j=lo; j < hi; j++)
      result += arr[j];
   return result;
}
```

A pseudocode FORALL loop is like a loop, except that it does all the iterations in parallel.

This slide adapted from: Sophomoric Parallelism and Concurrency, Dan Grossman

- Java does not have a construct like a *forall* loop but programs can be written to work as though they are implementing a *forall* loop
- The parallel coding example in pseudocode shows an array with 4 entries and a forall loop which completes all iterations at the same time
- The forall loop sums the array in particular range into partial sums which are then added together at the end
- An example of a concurrent operation is the insertion of integers into a Hash Table that are occurring at the same time
- We need to prevent the case where a duplicate key is generated at the same time and we have to perform two insertions at the same location in the table at the same time (prevent interleaving)

```
class Hashtable<K,V> {
   …
   void insert(K key, V value) {
      int bucket = …;
      prevent-other-inserts/lookups in table[bucket]
      do the insertion
      re-enable access to arr[bucket]
   }
   V lookup(K key) {
      (like insert, but can allow concurrent
       lookups to same bucket)
   }
}
```

concurrent programs use **synchronization** to prevent multiple operations from interleaving in a way that leads to incorrect results

This slide adapted from: Sophomoric Parallelism and Concurrency, Lecture 1, Dan Grossman

- Removing the sequential assumption creates both opportunities and challenges
- In the program, we need ways to divide and coordinate work (in Java it could be threads)
- Algorithms allow much more work to be done but it is a lot more work for the programmer (i.e. more throughput but more organisation required)
- Data structures need to support concurrent access
- Usually, parallel programs have an element of concurrency in them, for example, the code that the
- So if parallel computations need access to shared resources, then the concurrency needs to be managed

Race Condition

- A **race conditions** is an error in a program where the output and/or result of the process is unexpectedly and critically dependent on the relative sequence or timing of other events
- The idea is that events race each other to influence the output first
- Unintended race conditions are common in multithreaded programs
- In the case where multiple bakers share the same oven, a race condition would occur if two bakers simultaneously saw that the oven was empty and simultaneously set the condition *ovenEmpty* to false

```
if (ovenEmpty) {
   ovenEmpty=false;
   putCakeInOven();
   bakeCake();
   removeCakeFromOven();
   ovenEmpty=true;
}
```

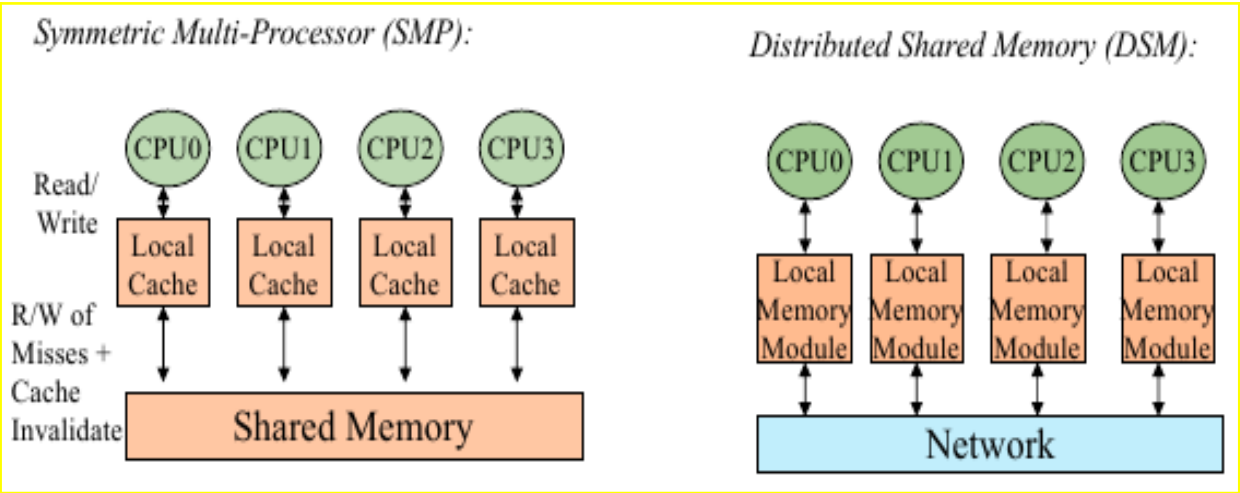
- The reason we obtain race conditions in concurrent programming is that it is nondeterministic
- In sequential programs, instructions are executed in a fixed order determined by the program and its input
- The execution of one procedure does not overlap in time with another. Because of this natural ordering, sequential programs are called **deterministic**
- However, in concurrent programs, computational activities may overlap in time and the subprogram executions describing these activities proceed concurrently – **non-deterministically**
- Concurrent computing is necessary for very complex programs like operating systems, web browsers, real-time systems, event-based implementations of GPU interfaces and multiuser games, chats and ecommerce
- The programs all handle multiple things at once and therefore are multithreaded
- If the computer has multiple processors the instructions form a number of processes/threads, equal to the number of physical processors, can be executed at the same time, sometimes referred to as parallel or real concurrent executions
- However, it is usual to have more active processes than processors
- **Timeslicing** refers to the technique where available processes are switched between processors
- Processor scheduling is managed by the operating system and the programmer has no control over it

B. Parallel and Concurrent Programming Model

- To write a parallel program, the programmer needs new primitives from a programming language or library that enable them to:
    + Run mutiple operations at once
    + Share data between operations
    + coordinate operations (synchronise operations)
- **Java** uses the **Shared Memory parallel programming model**

B.1 Shared Memory Model

- The shared memory model maps onto the shared memory physical architecture
- All memory is placed into a single (physical) address space
- Processors are connected by some form of interconnection network
- We assume that there is a single virtual address space across all of memory. Each processor can access all locations in memory
- A typical CPU has a local cache which it writes to and this cache interacts with the shared memory – **symmetric multi-processor** (SMP) architecture
- An alternative would be to make a cluster of separate machines which are connected in a network to behave like they are using a shared memory through local memory modules – **distributed shared memory** (DSM) architecture



- The performance of a program depends on the actual architecture of the machine
- CPUs “talk” to each other through the shared memory

- A **process** is represented by its code, data and the state of the machine registers
- Operating systems are units of resource allocation for both CPU time and memory
- The data of a process is divided into global variables and local variables which are organised as a **stack**
- Generally, each process in an operating system has its own address space which is an entirely separate entity
- A **thread** is a process given internal concurrency with multiple lightweight processes
- A process with multiple lightweight threads of control has multiple stacks
    - one for each thread
- Threads have private variables and shared variables
- Older operating systems used the Process Memory Model whereas newer operating systems used thread memory models
- The shared memory model has multiple explicit threads, i.e. threads run in parallel and are shared with the separate cores on a machine
- **Threads can**:
    + perform multiple computations in parallel
    + perform separate simultaneous activities
    + communicate easily and implicitly with each other through shared memory. Variables must be protected c correctly
- Implicit communication occurs when a CPU writes data to the shared memory for one thread and another thread can read the data without the programmer having to code an instruction

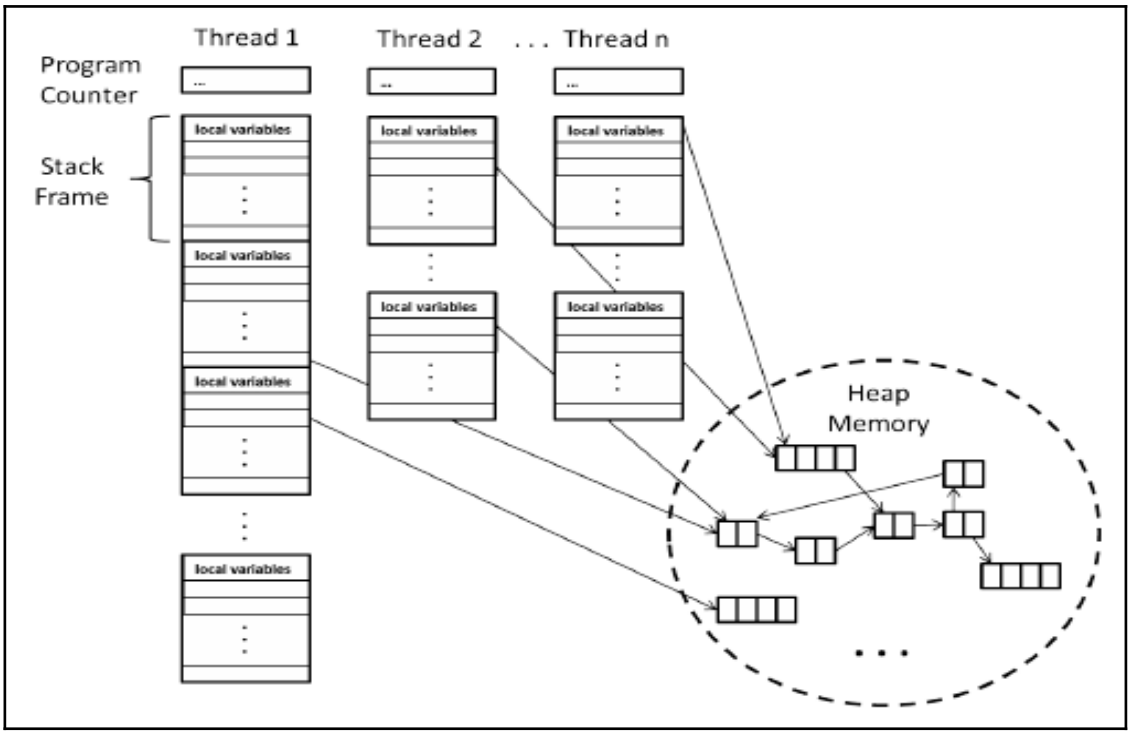
Sequential Program State Model vs Shared Memory Model

Sequential Program State Model	Shared Memory Model
One program counter	Multiple program counters for each thread
One call stack	Multiple call stacks for each thread
Objects	Shared objects
Static fields	Static fields of classes shared

- For the sequential program state model, the heap and the call stack located in different memory locations
- A sequential program has one program counter corresponding to the current statement executing

- The Objects in the sequential program state model are created by calling new. We call the memory that holds all the objects the *heap memory*

Key pieces of an executing multithreaded program including the heap memory



- In the threading shared memory model, each thread has its own program counter, call stacks and local variables
- All threads share one collection of Objects and **static** fields
- Static fields of classes are also shared by all threads
- Threads communicate through shared Objects (implicit communication)
- Shared Objects can be accessed at the same by different threads in the Shared Memory Programming Model
- Any Object can be shared but most Objects are not shared because sharing Objects increases program complexity and time costs
- The programmer creates threads and **scheduling** determines the order in which threads are run and on which core these processes are executed
- The programmer has no control over the sequence in which threads are run and which core will be used
- This property of threads is referred to as **asynchrony**, as threads are subject to sudden unpredictable delays such as:
    + Cache misses (short delays)
    + Page faults (long delays)
    + Scheduling quantum finished (really long delays)

- Modern architectures all have caches assigned to each CPU/core
- A cache is a section of very fast memory that only that core has access to
- CPU caches can have multiple levels and are primarily used to increase the performance of the CPU by avoiding reading from main memory which can be a very slow process
- A **cache miss** occurs when a thread is running on a CPU and attempts to access a variable that has not been stored to the cache and has to be called from shared memory
- Accessing data from shared memory imposes delays on processes
- Another source of delay concerning the cache is the **cache invalidation** which occurs when one thread running on a certain core has read from shared memory, a variable that has been changed by another core in the shared memory
- This causes a delay as the variable has changed and will require the core with the invalid variable to read the new value from the main memory
- A **page fault** occurs when a thread has to read a page from disk that was supposed to be in main memory
- A significantly time-costly delay occurs when the **scheduling quantum is finished**, i.e. when a running thread is removed from a CPU before it has completed its task
- Java uses this shared memory model for parallel computing however, there are other models such as the **message-passing model**
- The message-passing model considers processes that are running on machines that are essentially isolated and share packages of the data that needs to be shared to one another
- Integral to the message-passing model are the hand-shaking type of send and receive functions
- This model is typically used in HPC systems (high performance computing systems) which have a message-passing interface
- This model is more scalable and more suited to a cluster type architecture where separate machines are connected through a network
- Another common parallel programming model is the **map reduce model** which is a data parallelism concept from functional programming languages such as LISP
- Map reduce has primitives for things like apply function to every element of an array in parallel
- This model is basically a way of doing parallelism without having to worry about threads but focusing on what the CPU must do instead

- The programmer just writes the operations that describe how to map data (e.g. multiplication with an integer) and reduce data (e.g. an instruction to take the maximum)
- Details of the underlying parallelization are hidden from the programmer, provided you can express your program using the available primitives
- System then does all the parallelisation using multiple cores
- MapReduce was developed by Google and the programming model has since been adopted by many software architects

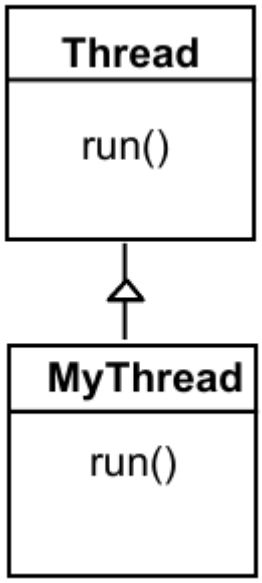
C. Java Threads

C.1.1 Introduction to Java Threads

- Recall that to write a parallel program, the programmer needs primitives from a programming language or library that enable you to run multiple operations at once, share data between operations and coordinate operations
- To **run multiple operations at once, Java has concurrent threads**
- To share data between operations, Java uses shared memory which all threads can access
- To **coordinate (synchronise) operations, Java has a range of synchronisation primitives,** as well as **thread-safe and concurrent classes**
- We will start with a simple primitive for threads to wait for each other
- Java provides both basic concurrency support in the Java programming language and the Java class libraries as well as high-level API
- Java is always multithreaded as the Java virtual machine executes as a process under the OS and supports multiple threads
- In Java, every program has more than one thread, however, we start with just one thread called the *main thread*
- The main thread creates multiple threads
- System threads perform garbage collection and signal handling, for example, handling the input from a mouse and keyboard
- Threads compile Java byecode into machine-level instructions
- Standard Java libraries use threads

C.1.2 Basic Threads in Java

- All threads are associated with an instance of the class **java.lang.Thread**
- There are two basic strategies for using **Thread Objects**:
  i. instantiate **Thread** each time the application needs to initiate an asynchronous task. This is the most common approach for concurrent applications
  ii. pass the application's task to an executor using the **java.util.Concurrent** package which will launch and manage threads. This is the typical approach for parallelisation
- Parallelisation requires a high-level API that manages a thread pool for large-scale applications that run on multi-processor and multi-core systems
- A **Thread Class** manages a single sequential thread of control. Threads may be created and deleted dynamically
- The **Thread Class** executes instructions from its method **run()**
- The actual code executed depends on the implementation provided for **run()** in a derived class



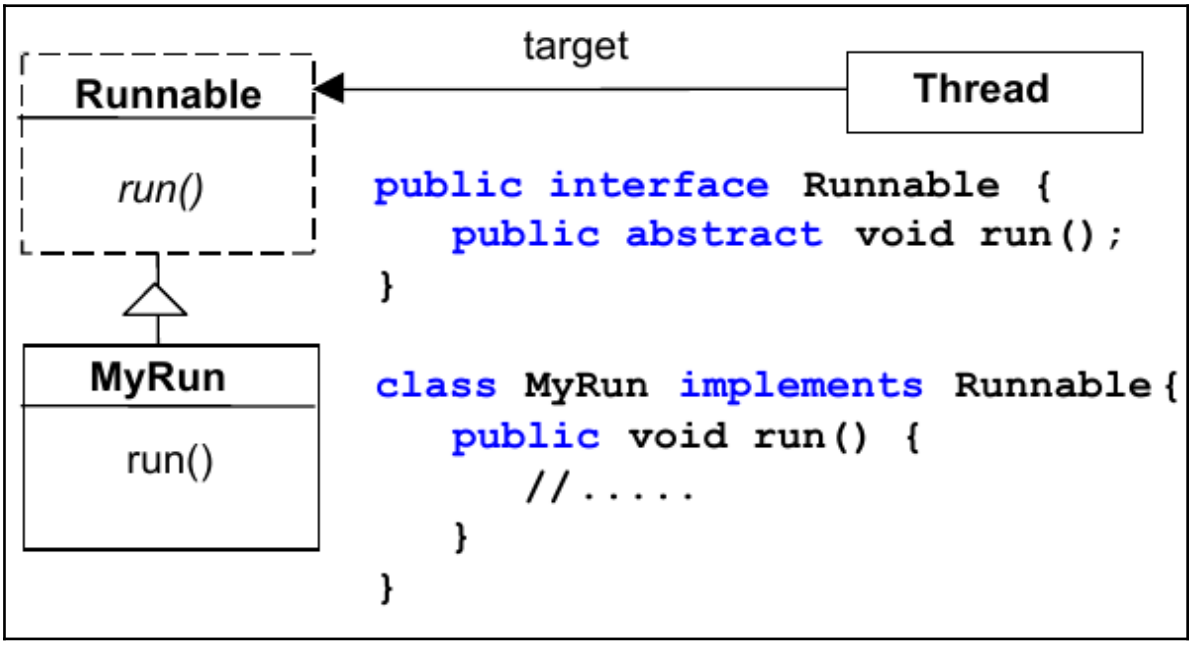

```
class MyThread extends Thread {
      public void run() {
            //......
      }
}
```

Creating a thread object:

```
Thread a = new MyThread();
```

- Since Java does not permit multiple inheritance, it is sometimes more convenient to implement the **run()** method in a class not derived from **Thread** but from the **Runnable** interface instead
- The Runnable interface defines a single method, **run()**, containing the code executed in the thread

Runnable Interface in Java



Creating a thread object:

```
Thread b = new Thread(new MyRun());
```

- The **Runnable Object** is passed to the Thread constructor
- In other words, there are two ways to create a basic thread in Java:
  - Extend/subclass the **Thread Class** (**java.lang.Thread**)
  - Implement the **Runnable Interface** in an Object and pass that to a Thread's **constructor** (**java.lang.Runnable**)
- Allocation and construction of a Thread Object do not cause the thread to **run**
- To get a new thread running:
  - Define a subclass C of **java.lang.Thread**, overriding run
  - Create an Object of class C
  - Call that Object's start method
    - Not run, which would just be a normal method call
    - **start** sets off a new thread, using **run** as its main
- Calling the **run** method of C would be a normal method call in the current thread

Sample Code C.1.1: HelloThread

```
public class HelloThread extends java.lang.Thread {
    private int i;
    HelloThread(int i) { this.i = i; }      Constructor

    public void run() {
        System.out.println("Thread " + i + " says hi");
        System.out.println("Thread " + i + " says bye");      This is what each thread does.
    }

    public static void main(String[] args) {
        for(int i=1; i <= 10; ++i) {
            HelloThread c = new HelloThread(i);
            c.start();
        }                                   This is what makes the thread run

}
```

- When this program runs, it will print 10 lines of output, some of which are:

  Thread 1 says hi
  Thread 8 says hi
  Thread 3 says hi

- **System.out.println()** is a **synchronised** method so it will always keep a line of output together
- This means that no other thread can interrupt this method and outputs such as

  Thread 13 Thread says 14 says hi hi ✗

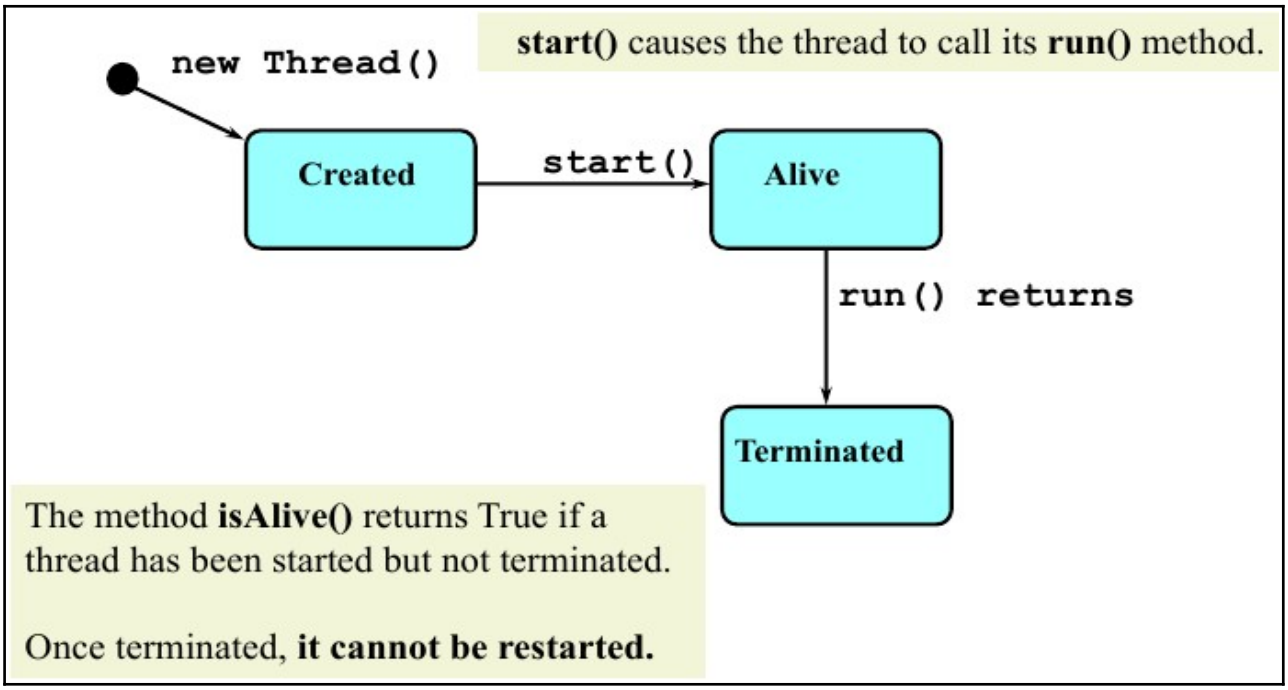
- We can also run create the Hello World thread using the **Runnable Interface**

Sample Code C.1.2: HelloInterface

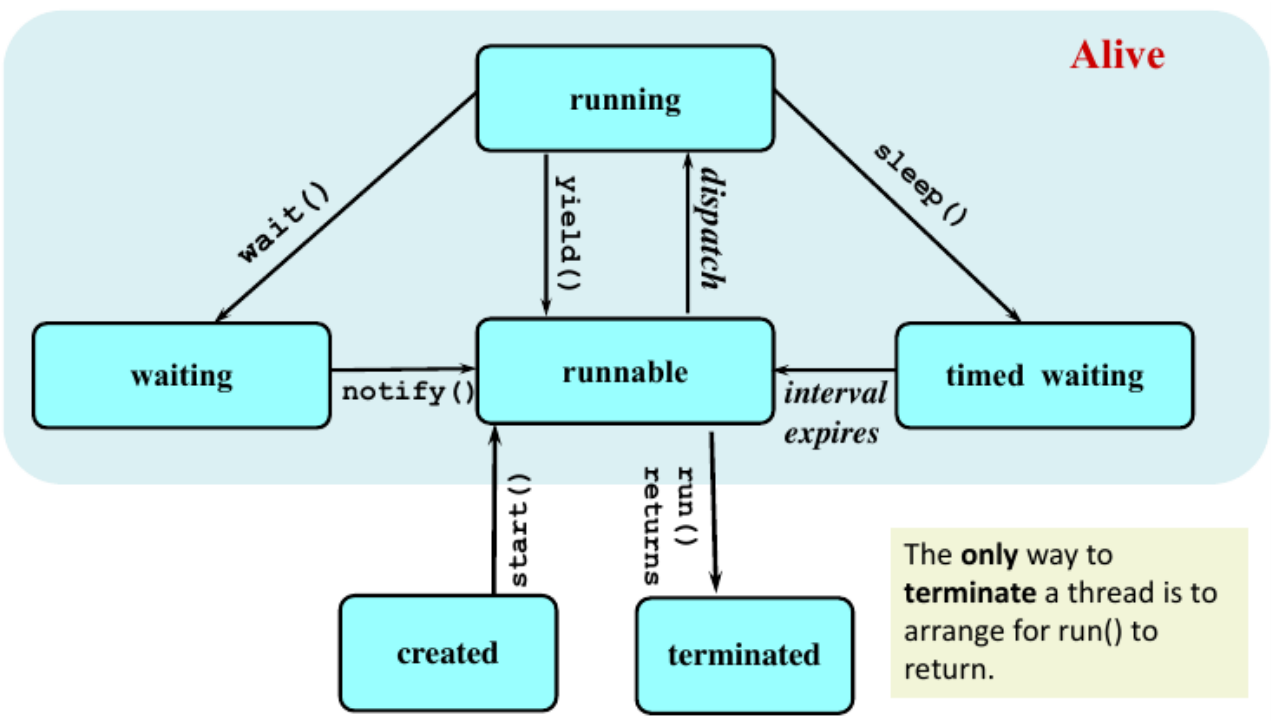
```
public class HelloInterface implements Runnable {
    private int i;
    HelloInterface(int i) { this.i = i; }

    public void run() {
        System.out.println("Thread " + i + " says hi");
        System.out.println("Thread " + i + " says bye");
    }

    public static void main(String[] args) {
        for(int i=1; i <= 10; ++i) {
            Thread c = new Thread(new HelloInterface(i));
            c.start();
        }
    }
}
```

- In the above code, we create and run a thread by passing the **HelloInterface** Object, implementing the **Runnable Interface**, to a **Thread** Object constructor and calling the **start** method of the **Thread Object**

An overview of the life-cycle of a thread as state transitions



- The most commonly used methods from the **Thread Class** include:
  - **start()** = starts the thread
  - **sleep(long** *millis***)** = pause the thread for the specified time in milliseconds
  - **yield()** = hint to the scheduler that the current thread is willing to yield its current use of a processor
  - **join()** = wait the current thread until the thread that called this method has terminated
- Once started, an alive thread has a number of substates shown in the block diagram below



- When a thread is created, it does not do anything until the **start** method is created
- The **start** method makes the thread *runnable* but it will not necessarily run
- In the context of the OS, runnable means that the thread is created and is placed in the *ready queue* as it waits for the Scheduler to allocate a CPU for the thread to run on
- Once running, the thread can **yield** to the Scheduler and stop running
- At this point, the Scheduler will probably move the thread off the CPU and make it runnable again or the thread will wait until it receives a message from another thread
- The **sleep** method can be used to make one thread slower than the others

- Since we have no control on how the OS Scheduler runs the threads in a program, it is not advisable to use the **setPriority()** method to run a thread in preference to another thread
- This method can be used to hint to the OS Scheduler about a certain ranking in the priorities of the threads
- **Thread priorities** increase platform dependence (OS and architecture dependence) and can cause liveness problems and should be avoided
- Liveness problems are issues with threads such as deadlocks where multiple threads wait on each other forever
- Most concurrent applications can use the default priority for all threads
- Although thread priorities exist in Java and many references state that the JVM will always select one of the highest priority threads for scheduling, this is not guaranteed by the Java language or the virtual machine specifications
- Priorities are ONLY hints to the Scheduler
- The only way to terminate a thread is to wait for **run() return**
- In the original design of the JVM, there were methods called **Thread.stop()** and **Thread.suspend** which were an attempt to stop threads safely but have now been deprecated
- Most of the time we allow threads to stop by running to completion
- Sometimes we want to stop threads sooner, e.g. when a user cancels an operation or when an application needs to shutdown quickly
- It is not easy to get threads to stop safely, quickly and reliably
- Java no longer provides any mechanism for forcing a thread to stop, instead, ask the thread to stop what it is doing with an **interrupt**
- The **java.util.Concurrent** extensive library is useful in concurrent and parallel programming and contains Executors, queues, timing, synchronizers, concurrent collections
- This library makes threading simpler, easier and less-error prone

Checkpoint C.1.3

Write down all the possible outputs of this code.

```
public class AThread extends Thread {
        public void run() {System.out.println("A"); }
}
public class BThread extends Thread {
        public void run() {System.out.println("B"); }
}
public class STest {
        public static void main(String[] args) {
                AThread threadA = new AThread();
                BThread threadB = new BThread();
                threadA.start();
                threadB.start();
                System.out.println("C");
        }
}
```

The above code could possibly output any of the 6 permutations:
ABC ACB BAC BCA CBA CAB

```
public class AThread extends Thread {
        public void run() {System.out.println("A"); }
}
public class BThread extends Thread {
        public void run() {System.out.println("B"); }
}
public class STest throws InterruptedException {
        public static void main(String[] args) {
                AThread threadA = new AThread();
                BThread threadB = new BThread();
                threadA.start();
                threadB.start();
                threadA.join();
                System.out.println("C");
        }
}
```

The above code includes a ***join*** synchronisation parallel programming method. The join on threadA ensures that 'A' is always printed before 'C'.

ABC BAC ACB

```
public class AThread extends Thread {
        public void run() {System.out.println("A"); }
}
public class BThread extends Thread {
        public void run() {System.out.println("B"); }
}
public class STest throws InterruptedException {
        public static void main(String[] args) {
                AThread threadA = new AThread();
                BThread threadB = new BThread();
                threadA.start();
                threadA.join();
                threadB.start();
                System.out.println("C");
        }
}
```

In the above code where we call a start on threadA and then a *join* on threadA, 'A' will always be printed first because threadB will not start until threadA is finished and 'C' will not print until threadA is finished. Therefore, there is only one possible outcome.

ABC

C.1.3 Java Executors for Parallel Programming

- In large-scale parallel applications, it makes sense to separate thread creation and management from the rest of the application
- **Executor Objects** in Java distribute tasks to worker threads in a thread pool
- Thread Pools (creates a lot of individual threads) reduce thread creation overhead:
    + allocating and deallocating many thread objects requires significant memory management
- The JVM decides how many threads are in a Thread Pool
- A task is divided into smaller tasks and the JVM assigns threads in the Thread Pool to complete each smaller task
- Java has three **Executor Interfaces:**
    + Executor
    + Shared Executor
    + Executor Service
- **Executor Interface** is the basic parent class for launching new tasks
- The **Executor Service Interface** adds features to manage tasks and is a sub-interface of the **Executor Interface**
- Of the **Executor Service Interface**, we will speak of the **Fork/Join** implementation
- The **Fork/Join** implementation was designed for **divide-and-conquer algorithms** which essentially take a big task and divide it into two
- *Merge Sort* is a divide-and-conquer algorithm since it looks sorts segments of an collection of items
- **Fork/Join** is also a *work-stealing* algorithm as idle threads can steal tasks from busy threads
- The ForkJoinPool is the main thread pool class
- To use this framework, create a ForkJoinPool and define a task which will be sent to the pool
- The task can either be:
    + *RecursiveAction* which has no return value
    + *RecursiveTask* which has a return value
- For image processing, we use *RecursiveAction*
- Below, we see an example of a Hello World program which uses *RecursiveAction*

Sample Code C.1.3: HelloMany

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.RecursiveTask;

// recursiveAction has no return value, recursiveTask returns a value
public class HelloMany extends RecursiveAction  {
    int greetings; // number of hellos
    int offset; // to know where to start which is otherwise difficult to do without offset

    HelloMany(int g, int start) {
      greetings=g;
      offset=start;
    }

    //compute in the ForkJoinPool framework is equivalent to run in the Thread Pool
    //the program needs to divide up the tasks for its multiple threads
    protected void compute(){
        if((greetings) <=1) { //only one task left, do it. This cutoff would be bigger for proper programs
            System.out.println("hello"+offset );//the thread will know its offset when its given its task to do
        }
          else {
              int split=(int) (greetings/2.0);//otherwise split greetings into half
              HelloMany left = new HelloMany(split,offset);  //first half
              HelloMany right= new HelloMany(greetings-split,offset+split ); //second half
              left.fork(); //give first half to new thread in the pool
              //left.join(); // what will this do if included?
              right.compute(); //do second half in this thread
              //right.fork would make the main thread redundant
          }
      }
    public static void main(String[] args) {
        HelloMany sayhello = new HelloMany(50,0); //the task to be done, divide and conquer
        ForkJoinPool pool  = new ForkJoinPool(); //the pool of worker threads
        pool.invoke(sayhello); //start everything running - give the task to the pool
      }
    }
```

- The above code will not execute correctly as a different number of threads will be run each time because of a race condition
- To fix this code, we need to add a **left.join()** after the **right.compute()**
- If the parent of a thread dies, the OS will kill all of its children threads. In this Fork/Join framework, each thread is creating a child lower down the framework

D. Parallel Programming In Java

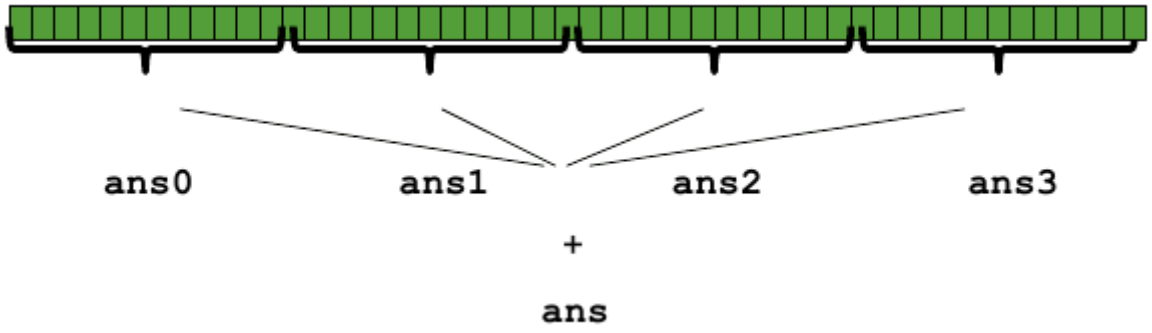
D.1 Parallel vs Sequential Programming in Java

- A common basic parallel programming illustration involves the summing of elements of a large array
- This illustration does not show the full capabilities of the framework as the threads are doing to little work
- We time the sequential/serial solution as a benchmark. Fill a big array with 1's and see how long it will take to sum the elements
- For assignment 1, we benchmark the size of the image, the size of the filter and the different architectures (nightmare)
- Do the sum twice and time each (to check for cache effects)

Sample Code D.1.1.1: Serial Solution for Benchmarking

```
package SerialSum;
public class SumAll {
    static long startTime = 0;

    private static void tick(){
        startTime = System.currentTimeMillis();
    }
    private static float toc(){
        return (System.currentTimeMillis() - startTime) / 1000.0f;
    }

    public static void main(String[] args) {

        int ans=0;
        int max =100000000;

        int [] arr = new int[max];
        for (int i=0;i<max;i++) {  // for checking purposes
            arr[i]=1;
        }

        //first run
        tick();//start time
        for (int i=0;i<max;i++) {  // for checking purposes
            ans += arr[i];
        }
        float time = toc(); //execution time given by time when solution is done minus start time
        System.out.println("Run1 took "+ time +" seconds for " + max + " additions");
        System.out.println("Sum is:" + ans);

        //second run
        ans=0;
        tick();
        for (int i=0;i<max;i++) {  // for checking purposes
            ans += arr[i];
        }
        time = toc();
        System.out.println("Run2 took "+ time +" seconds for " + max + " additions");
        System.out.println("Sum is:" + ans);
    }
}
```

- In the trivial parallel programming version of summing elements of a big array, the idea is to have 4 threads simultaneously summing ¼ of the array
- Each thread will find the partial sum and their results will be added together to finish the program execution



- Recall that a thread must call the **start()** method to become a runnable. In the sample code below, the result will be 0, which is wrong, as the threads in the array of threads have only been created using the **constructor** but have not started running

Sample Code D.1.1.2: Demonstrating exclusion of start method

```
package Attempt1;

public class SumThread extends Thread {
	int lo; // start of array segment
	int hi; // end of array segment
	int[] arr;

	int ans = 0; // result

	SumThread(int[] a, int l, int h) {
	  lo=l; hi=h; arr=a;
	}

	public void run() { //override must have this type
	  for(int i=lo; i < hi; i++)
	    ans += arr[i];
	}

}
```

```
//WRONG <=> has race condition
//WRONG <=> threads do not call start() method
package Attempt1;
public class SumAll {
    static int sum(int[] arr, int numTs)  {
        int ans = 0;
        SumThread[] ts = new SumThread[numTs]; //creates an array of threads with numTs indices
        // the elements in the array are actually references to memory
        // so a constructor must be called each time
        for(int i=0; i < numTs; i++){  // do parallel computations
            ts[i] = new SumThread(arr,(i*arr.length)/numTs,
                                  ((i+1)*arr.length)/numTs);// give each thread range of sum
            // each thread gets the same array and starts summing from lo to hi
        }
        for(int i=0; i < numTs; i++) { // combine results
          ans += ts[i].ans;// sum the answers after threads have run and return the answer
        }
        return ans;
      }
    public static void main(String[] args) {
        int max =10000000;
        int noThreads =4;
        int [] arr = new int[max];
        for (int i=0;i<max;i++) {  // for checking purposes
            arr[i]=1;
        }
        int sumArr = sum(arr,noThreads);
        System.out.println("Sum is:");
        System.out.println(sumArr);
    }
}
```

- A thread cannot start itself so we need to initialise the thread and then call the **start()** method
- In the second attempt of the problem below, we change the **SumAll** class to **start()** each thread after creating each one

Sample Code D.1.1.3: Demonstrating race condition for excluding join

```
static int sum(int[] arr, int numTs) throws InterruptedException {
        int ans = 0;
        SumThread[] ts = new SumThread[numTs];
        for(int i=0; i < numTs; i++){
            ts[i] = new SumThread(arr,(i*arr.length)/numTs,
                                  ((i+1)*arr.length)/numTs);
            ts[i].start();  //start, not run
        }
        for(int i=0; i < numTs; i++) {
          ans += ts[i].ans;
        }
        return ans;
      }
```

- The *lo*, *hi* and *arr* fields are written by the main thread and read by the helper threads
- The *lo* and *hi* fields are written once for each thread however, each *ans* field is written by each helper thread and read by the main thread
- Therefore, we have a race condition on ts[i].*ans* field which is read by the main thread
- We may be attempting to read *ans* from thread *i* which may not be done
- We need to implement the **join** method on each thread in the array to ensure that we only access the *ans* field of each thread when they are done

Sample Code D.1.1.3: Partially correct solution using join

```
static int sum(int[] arr, int numTs) throws InterruptedException {
    int ans = 0;
    SumThread[] ts = new SumThread[numTs];
    for(int i=0; i < numTs; i++){
        ts[i] = new SumThread(arr,(i*arr.length)/numTs,
                              ((i+1)*arr.length)/numTs);
        ts[i].start();  //start, not run
    }
    for(int i=0; i < numTs; i++) {
      ts[i].join(); //wait for each thread to be terminated
      ans += ts[i].ans;
    }
    return ans;
  }
```

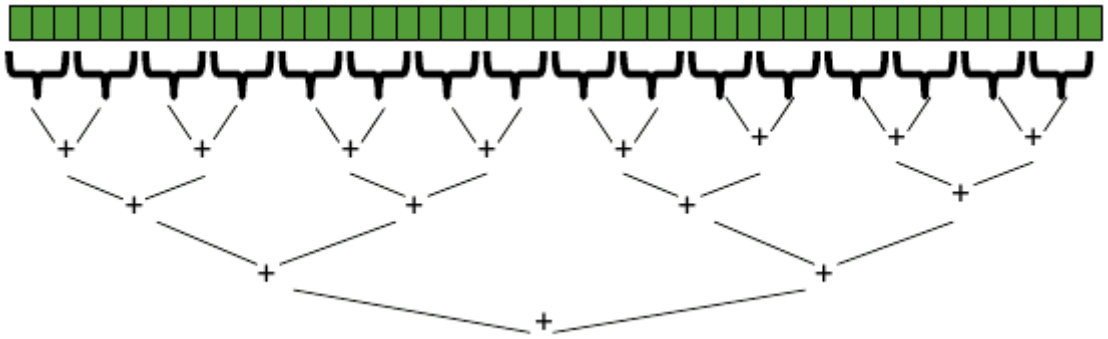
- The **join()** insists on each helper thread to terminate before the main thread can read the *ans* field
- This method requires that we throw an **InterruptedException**
- Now that the race condition on *ans* has been eliminated, we can find the execution time of the program by defining a start time and subtracting that start time from the current time when the result has been returned

Sample Code D.1.1.4: Correct parallel version with execution time

```
package Attempt3timed;
public class SumAll {
    static long startTime = 0;
    static long runTime = 0;

    private static void tic(){
        startTime = System.currentTimeMillis();
    }
    private static void toc(){
        runTime = (System.currentTimeMillis() - startTime) ;
    }

    static int sum(int[] arr, int numTs) throws InterruptedException {
        int ans = 0;
        SumThread[] ts = new SumThread[numTs];
        for(int i=0; i < numTs; i++){
            ts[i] = new SumThread(arr,(i*arr.length)/numTs,
                                  ((i+1)*arr.length)/numTs);}
        tic(); //start
        for(int i=0; i < numTs; i++) {
            ts[i].start();  //start threads working
        }
        for(int i=0; i < numTs; i++) {
          ts[i].join();
          ans += ts[i].ans;
        }
        toc(); //end
        return ans;
      }

    public static void main(String[] args) {
        int max =100000000;
        //int noThreads =10;
        int noThreads = Runtime.getRuntime().availableProcessors(); // get from system
        int [] arr = new int[max];
        for (int i=0;i<max;i++) {
            arr[i]=1;
        }
        try {

            int sumArr = sum(arr,noThreads);
            System.out.println("Sum is:"+ sumArr);
            System.out.println("Run1 took "+ runTime/ 1000.0f +" seconds for " + max + " additions with "+ noThreads + " threads");

            sumArr = sum(arr,noThreads);
            System.out.println("Sum is:"+ sumArr);
            System.out.println("Run2 took "+ runTime/ 1000.0f +" seconds for " + max + " additions with "+ noThreads + " threads");

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- Alternatively, we can use the **Fork/Join** framework which is more straightforward to implement
- The **Fork/Join** framework is a **divide-and-conquer** algorithm which uses parallelism specifically for recursive calls
- **Result-combining** is also done in parallel and is more efficient than the serial version of the code
- If you have enough processors, total time is height of the tree, i.e. O(log *n*), exponentially faster than the serial code's O(*n*)
- In this framework, the divide-and-conquer algorithm aims to break down a process into individual threads until it is small enough to execute on one core to complete very quickly

- At the end of the process, the main thread sums up two numbers from two threads, however, these two numbers are a result of the framework finding the sums of the smaller threads



- Recall that for the **Fork/Join** framework, we define a ForkJoinTask which is a RecursiveAction type which has no return value or RecursiveTask type which has a return value
- These tasks are given to a ForkJoinPool which contains multiple threads to complete these tasks recursively
- As in the example in Sample Code C.1.3, we can create the ForkJoinPool as a **static final** field, e.g.

```
static final ForkJoinPool fjPool = new ForkJoinPool();
```

- We use **static final** because the ForkJoinPool is a class level pool and is shared by each thread
- **ForkJoinPool()** creates a ForkJoinPool with typically the same number of threads (parallelism) as **Runtime.availableProcessors()** (the number of virtual processors available on the machine)
- The parallelism can be specified by the program for algorithms where it may be beneficial to have fewer threads or more threads
- The recommended way of creating a pool is to use **commonPool()** to create the default ForkJoinPool
- This static method creates a pool with parallelism equal to the **Runtime.availableProcessors()** - *1* so that the main thread can run on the remaining pool
- Another benchmarking parameter that can be implemented is the number of threads in the ForkJoinPool to see how many threads give optimum performance
- Sample code C.2.1.5 shows the final version of the summing program implementing the ForkJoin framework

Sample Code D.1.1.5: Final version with SumArray using ForkJoin framework

RecursiveTask definition class

```
package ForkJoinDC;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class SumArray extends RecursiveTask<Integer>  {// task defines divide-&-conquer algorithm
    int lo; // arguments
    int hi;
    int[] arr;
    static final int SEQUENTIAL_CUTOFF=5000000;
    int ans = 0; // result
    SumArray(int[] a, int l, int h) {
      lo=l; hi=h; arr=a;
    }
    //compute function is equivalent to the run command with normal threads
    protected Integer compute(){// return answer - instead of run
        if((hi-lo) < SEQUENTIAL_CUTOFF) {// check if we are below the sequential cut-off
            int ans = 0;
            for(int i=lo; i < hi; i++)
              ans += arr[i];
            return ans;
        }
        else {//divide-and-conquer
            SumArray left = new SumArray(arr,lo,(hi+lo)/2);//first half of array
            SumArray right= new SumArray(arr,(hi+lo)/2,hi);//second half of array
            left.fork(); //recursively execute the compute method with the parameters in
            // that array in another thread. This is where the next thread gets created
            int rightAns = right.compute(); // this is a RecursiveTask so it returns a value
            //we run the right part in the current thread for efficiency reasons
            int leftAns  = left.join();   // this is the result from the sum of the left hand side
            return leftAns + rightAns;      // the order in which these methods are executed is very
            // to avoid a race condition
        }
    }
  }
```

Application class creating and running ForkJoinPool

```
package ForkJoinDC;
import java.util.concurrent.ForkJoinPool;
import  java.util.concurrent.ForkJoinTask;
import java.util.concurrent.RecursiveTask;

public class SumAll {//creates a ForkJoinPool
    static long startTime = 0;
    private static void tick(){
        startTime = System.currentTimeMillis();
    }
    private static float toc(){
        return (System.currentTimeMillis() - startTime) / 1000.0f;
    }
    static final ForkJoinPool fjPool = new ForkJoinPool();
    static int sum(int[] arr){
      return fjPool.invoke(new SumArray(arr,0,arr.length));//ForkJoinPool executes a SumArray task
      //invoke is equivalent to start for normal threads
    }
    public static void main(String[] args) {//creates ForkJoinPool and give the task to the pool
                                            //using the sum() method defined above
        int max =100000000;
        int [] arr = new int[max];
        for (int i=0;i<max;i++) {
            arr[i]=1;
        }
        tick();
        int sumArr = sum(arr);
        float time = toc();
        System.out.println("F/J run1 took "+ time +" seconds for " + max +
            " additions with seq. cutoff of " +SumArray.SEQUENTIAL_CUTOFF);

        System.out.println("Sum is:");
        System.out.println(sumArr);
        tick();
        sumArr = sum(arr);
        time = toc();
        System.out.println("F/J run2 took "+ time +" seconds for " + max + " additions with seq. cutoff of "
        +SumArray.SEQUENTIAL_CUTOFF);

        System.out.println("Sum is:");
        System.out.println(sumArr);
    }
}
```

- To get the sequential cut-off for tasks, we could theoretically divide down to single elements, do all the result-combining in parallel and get optimal speedup
- This could be very fast if creating and passing variables to the threads is not time consuming
- In practice, there is a point where the fork costs more than the calculation so we use a sequential cutoff which depends on the algorithm but is typically 500-1000
- Exactly like quicksort switching to insertion sort for small subproblems but more important here
- It is not recommended to create two recursive threads; create one and send the other to the framework to complete
- The stopping condition for the recursion occurs when we reach the sequential limit
- When running the benchmark, we may see slow results before the JVM reoptimises the library internals
- We say that the framework need to warm up
- Fork/Join is really useful when:
    + when you are doing the parallel computation many times
    + when threads have a lot to do
    + when threads have different amounts of work to do - load imbalance
- The Fork/Join framework provides nearly ideal speedups for nearly any Fork/Join program on commonly available 2-way, 4-way, and 8-way SMP machines

D.2 Parallel Performance

- The more work that can actually be divided amongst the threads, the better the speed-up will actually be
- The example where the elements of an array are summed up actually takes more work and therefore costs more in running the program, i.e. creating the pool, synchronisation and other parallelism related tasks take more time than the actual computations
- This is why it is not the best idea to implement the F/J framework in this case
- To justify the existence of parallelism, parallel programs have to be both accurate and efficient
- To demonstrate correctness, benchmark an algorithm against the serial version to show that it produces the same results across a range of input
- To demonstrate efficiency (speed-up), benchmark an algorithm against the serial version, to show that it is (ultimately) faster

D.2.1 Work and Span

- Let T_p be the run-time with P processors
- There are two key measures of run-time:

  + *Work* => how long it would take 1 processor (time T_1), i.e. the serial time
  + *Span* => how long it would take infinity processors i.e. time T_∞

- For divide-and-conquer algorithm, the span is O(log n) for summing an array
- So span is basically how good you can get the parallel program to work
- The span is also called the critical path length, i.e. the number of operations that need to be executed even if we had an infinite number of processors
- The *Work Law* and *Span Law* give the constraints on the run-time T_p

Work Law

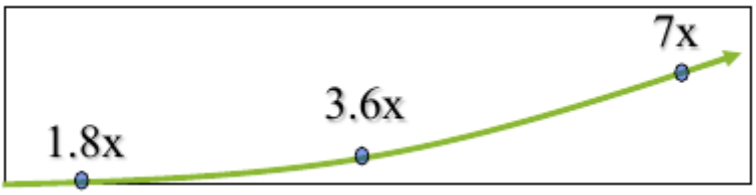
$$T_P >= T_1 / P$$

Span Law

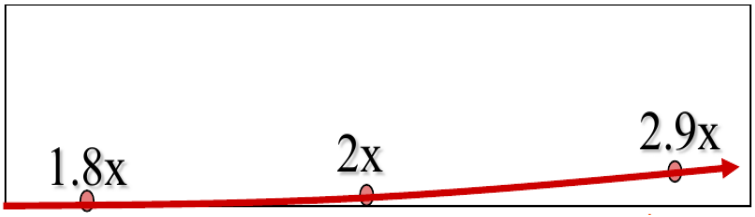
$$T_P >= T_\infty$$

- Work Law: each processor executes at most 1 instruction per unit time, and hence P processors can execute at most P instructions per unit time, therefore to do all the work on all the processors, it must take at least T_1/P time
- Span Law: a finite number of processors cannot outperform an infinite number of processors, because the infinite-processor machine could just ignore all but P of its processors and mimic a P-processor machine exactly
- This is applicable if we ignore memory-hierarchy issues
- Speed up is the time it takes for 1 process to complete over time for P processes
- Perfect linear speed-up is related to P since doubling the cores, for example, would half the execution time. This corresponds to the *Work Law*
- Scalability is an easier way to achieve goal. An algorithm is scalable if the speed-up increases at least linearly with the problem size
- Running time is inversely proportional to the number of processors used
- Still hard to get linear scalability in practice because most algorithms reach an optimal level of performance and the deteriorate
- The point is that we cannot speed-up that is greater than the number of available cores

Multicore Scaling Process



Real-World Scaling Process



- The multi-core scaling process was governed by Moore's Law as the number of cores in computers increased exponentially over time
- However, the real-world scaling process is constrained by other factors that may change the scalability as algorithms become more demanding
- **Parallelisation overhead** refers to the amount of time required to coordinate parallel tasks, as opposed to doing useful work, e.g. starting threads, stopping threads, synchronisation

D.2.2 Amdahl's Law

- Parallel computing became more pursued thoroughly in the 90's
- Before this, the dramatic increase in uniprocessor speed did not give incentive to pursue parallelism
- *Amdahl's Law* is a mathematical theorem demonstrating the diminishing returns of adding more processors
- Typically, only some parts of programs parallelise well, e.g. maps/reductions over arrays (call this the parallel fraction, p)
- Some parts are inherently sequential, e.g. reading a linked list, getting input, doing computations where each needs the previous step
- Amdahl stated that the speed-up is given by

$$\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}$$

  where n is the number of processors and p is the parallel fraction
- The sequential fraction is given by the term $1\text{-}p$ in the equation above
- In terms of work and span, Amdahl's Law implies that the serial portion of the program restricts the minimum span, thus

$$\mathbf{T}_\infty > (1-p)\, T_1$$

- So the maximum possible speed-up is

$$T_1 / \mathbf{T}_\infty < 1/(1-p)$$

- In the equation for Amdahl's Law, we can see how this is true as the term p/n tends to goes to zero as the number of processors increases
- For example, given 10 processors and a program with 60% parallel and 40% sequential parts, the closest we can get to 10-fold speed up is

$$\text{Speedup} = \frac{1}{1 - 0.6 + \frac{0.6}{10}}$$

$$\text{Speedup} = 2.17$$

- When we increase the portion of the program which is parallel to 90%, then we get closer to 10-fold speed-up

$$\text{Speedup} = \frac{1}{1 - 0.9 + \frac{0.9}{10}}$$

$$\text{Speedup} = 5.26$$

- This law implies that there will not be much benefit from increasing the number of cores too much as the speed-up is restricted by the part of the program which is sequential
- We only get exceptional speed-ups if we're less than 1% sequential and up to 5% sequential in some cases
- If a program is 33% percent sequential, then the maximum speed-up cannot exceed 3%
- To get at least 100x speed-up from 256 processors, we need the ratio of the program that is sequential to not exceed 0.0061, i.e.

$$\frac{1}{\left(S + \frac{(1-S)}{256}\right)} \geq 100$$

$$1 \geq 100\,S + \frac{100}{256}(1-S)$$

$$\left(100 - \frac{100}{256}\right) S \leq 1 - \frac{100}{256}$$

$$\Rightarrow S \leq 0.00612$$

where S is the ratio of the program that is sequential.

- In other words, 0.612% of the program should be sequential to achieve 100x speed-up with 256 processors
- Amdahl's Law suggests that it is not possible to speed-up an application indefinitely by using a large number of processors
- For many, this observation implied that parallel programming had no future
- Historically, this was correct as the complexity of the problems being dealt with was small for parallel computing so it was not going to be used efficiently
- The assumption that Amdahl's Law imposed did not hold true
- The ratio of T_S (serial execution time) to T_P (parallel execution time) is not constant for the same program
- The ratio varies with the size of the problem. Typically, T_P grows faster than T_S
- This is known as the *Amdahl effect* which states that the computational problem grows in size through the growth of parallel components

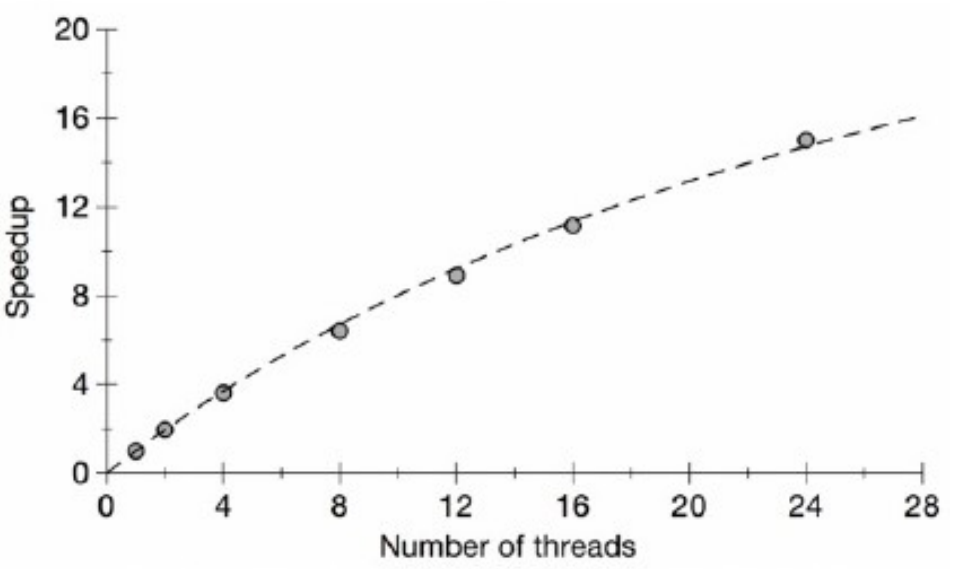
Scalability

- **Strong scaling**:
    + Fixed problem size
    + defined as how the solution time varies with the number of processors for a **fixed total problem size**
    + e.g. same task, double the number of processors
- **Weak Scaling**:
    + Fixed execution time
    + defined as how the solution time varies with the number of processors for a **fixed problem size per processors**
    + e.g. double the task while doubling the number of processors
- Scaling efficiency refers to the ratio of the run-time on one processor to the run-time on P processors

$$\eta_{scaling} = \frac{T_1}{T_P} \times 100$$

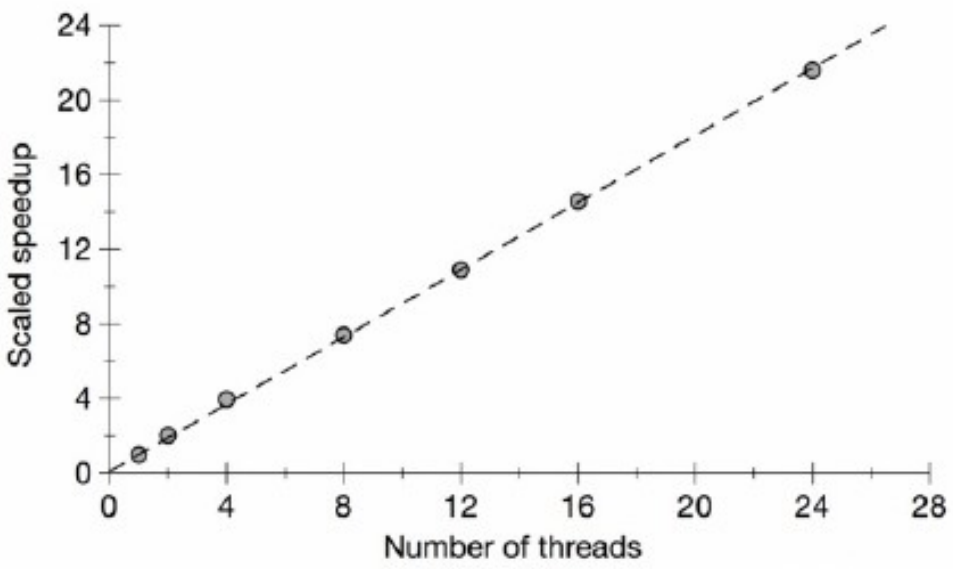
- With strong scaling, the problem size stays the same as the number of processors changes

Strong Scaling



- With weak scaling, the problem size doubles as thread size doubles

Weak Scaling



Checkpoint D.2.1

You are working on a program to analyse Facebook data. Your current implementation takes 15 minutes to run on a 4-core machine. The implementation takes 5 minutes with the sequential processing and the rest of the time is spent processing on all four processors (assume that it is perfectly linearly parallel). You want your code to run faster, should you hire someone to implement an alternative (more complicated) algorithm that promises 0.5 minutes sequential processing time or should you spend the same amount of money buying a 16 processor machine.

Soln:

We need to calculate the speed-up,

$$\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}$$

where the term (1-p) represents the fraction of the program which sequential and p is the fraction of the code that is parallel. We are given $n_1 = 4$, $n_2 = 16$. The sequential portion of the code is 5 minutes/15 minutes = 1/3 which means that the parallel portion is 2/3. Therefore, when using 4 cores, we obtain a speed-up of

$$Speed-up = \frac{1}{(\frac{1}{3}) + \frac{(\frac{2}{3})}{4}} = 2$$

If we change the number of cores to 16 cores, we obtain a speed-up of

$$Speed-up = \frac{1}{(\frac{1}{3}) + \frac{(\frac{2}{3})}{16}} = 2.67$$

i.e. squaring the number of processors increases the speed up by 33.5% percent. Lastly, if we reduce the fraction of the program that is sequential to run in 0.5 minutes, then the speed-up will

be change. The sequential fraction is 0.5 minutes/10.5 minutes = 1/21

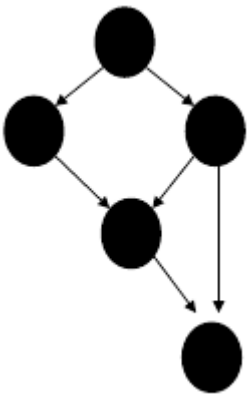
$$Speed-up=\frac{1}{(\frac{1}{21})+\frac{(\frac{20}{21})}{4}}$$
$$=3.5$$

Therefore, it would be better to hire the person promising a 0.5 minute sequential time as it gives the highest speed-up of 3.5.

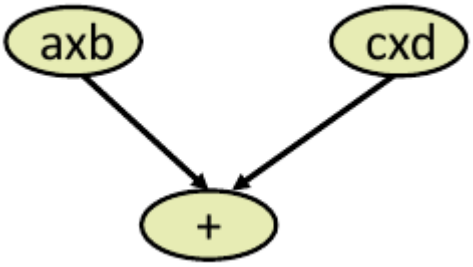
E. Parallel Programming Algorithms

E.1.1 Directed Acyclic Graphs for Parallelisation

- To visualise a program and determine the extent of parallelisation, we express the program execution as a directed acyclic graph (DAG) (also known as cost graph)



- The nodes in the graph represent pieces of work while the edges represent the flow of work where the source node must complete its work before the destination starts
- The longest path of dependencies indicates the sequential limit or **span**, i.e. T_∞
- A **fork** ends a node and makes two outgoing edges. The two outgoing edges could be a new thread or a continuation of the current thread
- A **join** ends a node and makes a node with two incoming edges
- The **work** of the program is represented by the total number of nodes, i.e. T_1
- The span which is represented by the longest path is called also known as the critical path of the program
- The span of the DAG shown above is 4 and the work is 5
- Given the expression a*b + c*d, we would need to perform the multiplication separately and then add them together in the same thread

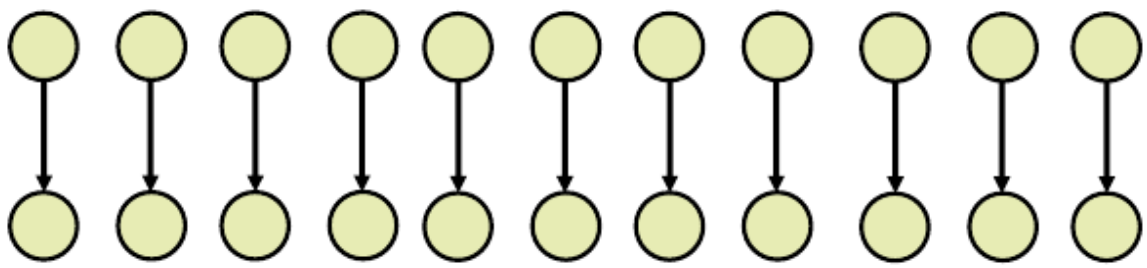


- The set of instructions forms the vertices (nodes) of the DAG
- The graph edges indicate the dependencies between instructiosn

- We say that an instruction *x* **precedes** an instruction *y* if *x* must be complete before *y* can begin
- The aim of parallel algorithms is to decrease the span without increasing work too much
- The point at which adding more processors no longer has an effect on the speed-up of a program depends on the span
- Parallelism can also be defined as the ratio of work to span, i.e. T_1/T_∞

E.1.2 Embarrassingly Parallel Algorithms

- An **embarrassingly parallel algorithm** is ideal and occurs when a computation that can be divided into a number of completely separate tasks, each of which can be executed by a single processor
- We do not need threading libraries or techniques but can write completely separate programs to perform each task
- Examples of embarrassingly parallel programs include:
  + element wise linear algebra (addition, scalar multiplication)
  + Image processing (shift, rotate, clip, scale)
  + Monte Carlo simulations
  + Encryption, compression
- The DAG for an embarrassingly parallel algorithm is shown below where each process is separated from the other and includes one two nodes connected by one edge



- In this case, each thread goes through each bi-node process without the requirements for a join or fork. Another example of an embarrassingly parallel DAG could be a series of isolated nodes

Examples E.1.2: Embarrassingly Parallel Algorithm Applications

i. Image processing

Low-level image processing uses the individual pixel values to modify the image in some way. Image processing operations can be divided into point-processing, local operations and global operations.

Within an image, **point-processing** is entirely embarrassingly parallel as the output produced is based on the value of a single pixel, e.g. the Mandelbrot set. However, a join is required at the end of point-processing so that the master thread is not killed before the sub-threads are finished running. Processing separate images is always embarrassingly parallel.

Local operations are partially embarrassingly parallel where we are calculating the value of the pixel based on a group of neighbouring pixels.

Global operations are not embarrassingly parallel as the output is produced based on all the pixels of the image.

ii. Monte Carlo Methods

The basis of Monte Carlo methods is the use of random selections in calculations that lead to the solution of numerical and physical problems, e.g. modelling Brownian motion, molecular modelling and forecasting the stock market. Each calculation is independent of the others and hence embarrassingly parallel.

For successful Monte Carlo simulations, the random numbers must be independent of each other. Developing random number generator

algorithms and implementations that are fast, easy to use, and give good quality pseudo-random numbers is a challenging problem. Developing parallel implementations is even more difficult.

The Mersenne Twister is a pseudo-random number generator algorithm developed by Matsumoto and Nishimura. This algorithm has good distribution properties, long period and efficient use of memory and high performance. This is the default algorithm used in Python, Julia and Matlab, however, it is not the default algorithm used in C or Java.

A good random number generator may be more effective and easier to implement for serial programs than it is for programs with high parallelisation. It is vital that there are no correlations between the random number streams on different processors, i.e. it is not ideal for one processor to repeat part of another processor's sequence.

The simplest solution is to have many simultaneous Mersenne twisters processed in parallel. Even very different initial state values do not prevent correlated sequences by generators sharing identical parameters. The *dcmt* is a special offline library used for dynamic creation of Mersenne Twisters parameters. The library accepts 16-bit thread id as one of the inputs and encodes this value into the Mersenne Twister parameters on a per-thread basis, so that every thread can update the twister independently, while still retaining good randomness of the final output.

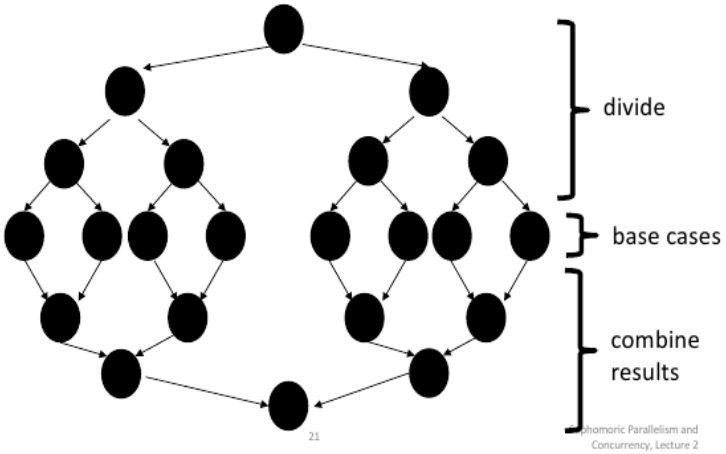
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf

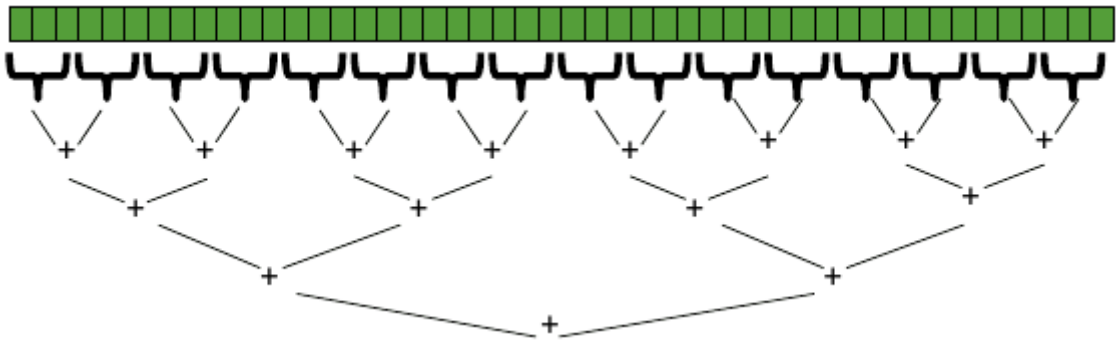
In Java, the standard pseudo-random number generator used is found in the **java.util.Random** library which is thread-safe but can have poor performance in a multi-threaded environment due to contention when multiple threads share the same Random instance. The alternative that does not have the same issues as the standard random number generator is the **ThreadLocalRandom** class which is isolated to the current thread. It is also initialised with an internally generated seed that may not otherwise be modified. When applicable, use of ThreadLocalRandom rather than shared Random objects in concurrent programs will typically encounter much less overhead and contention.

E.2 Divide-and-Conquer Parallel Algorithms

- Recall that the divide-and-conquer parallel algorithm divides problems into sub problems that are of the same for as the larger problem
  - i. Divide instance of problem into two or more smaller instances
  - ii. Solve smaller instances recursively
  - iii. Obtain solution to original (larger) instance by combining these solutions
- Recursive subdivision continues until the grain size of the problem is small enough to be solved sequentially
- Some examples of where the divide-and-conquer algorithm is applied include:
  - + finding the maximum or minimum element in an array
  - + is there an element satisfying some property
  - + left-most element satisfying some property
  - + corners of a rectangle containing all points (a “bounding box”). Join corners like knitting a the corners of a chequered jersey
  - + counts, e.g. number of strings that start with a vowel
    - this is just summing with a different base case
- Divide-and-conquer maps and reductions use **join** and **fork** in a basic way in relations to the flexibility they possess
- These are used to recursively create new nodes
- At each point where nodes recombine, i.e. where there are two edges going into a node, there is a **join**
- The base case corresponds to the point where the program has reached the sequential cut-off

- The span of the divide-and-conquer algorithm is equal to the heigh of the tree and the work is equal to the number of nodes in the tree
- Maximum speed-up is $n/\log n$, work over span, speed-up of the divide-and-conquer algorithm grows exponentially





- **Reduction** operations produce a single answer from a collection via an associative operator, e.g. reduce an array into a smaller collection
- A median is not a reduction operation as you do not have an associative operator
- Recursive results do not have to be single numbers or strings. They can be arrays or objects with multiple fields, for example, Histogram of test results is a variant of sum where an array of bar heights is returned
- Some things within the divide-and-conquer algorithm
- A **map** operates on each element of a collection independently to create a new collection of the same size

- A map is similar to a reduction in that it reduces a collection into smaller size however, a map performs operations on each element of the collection
- For example, for vector addition, we are adding two vectors together and obtaining a vector of the same size. The condition here is that the vectors that are being added must have the same cardinality
- We can perform vector addition using the Fork/Join framework very easily. The pseudocode is shown below

```
int[] add(int[] arr1, int[] arr2) {
  assert(arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  FORALL(int i=0; i < arr1.length; i++)
    ans[i] = arr1[i] + arr2[i];
  return ans;
}
```

Sample Code E.2.1: VectorAddition

```
class VecAdd extends RecursiveAction {
  int lo; int hi; int[] res; int[] arr1; int[] arr2;
  VecAdd(int l,int h,int[] r,int[] a1,int[] a2){ … }
  protected void compute(){
    if(hi – lo < SEQUENTIAL_CUTOFF) {
          for(int i=lo; i < hi; i++)
        res[i] = arr1[i] + arr2[i];
    } else {
      int mid = (hi+lo)/2;
      VecAdd left = new VecAdd(lo,mid,res,arr1,arr2);
      VecAdd right= new VecAdd(mid,hi,res,arr1,arr2);
      left.fork();
      right.compute();
      left.join();
    }
  }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  fjPool.invoke(new VecAdd(0,arr.length,ans,arr1,arr2);
  return ans;
```

from: Sophomoric Parallelism and Concurrency, Lecture 2 26

- In sample code E.2.1 on the previous page, we create a ForkJoinPool
- We use the **assert()** method to ensures that the vectors (arrays) are of the same dimensions. If the assertion is not true, the program will exit and give an output indicating where the it exited and why
- Recall that in this framework, **compute** splits the vector into left and right while the sequential cut-off has not been reached
- Even though there is no result-combining for maps in the Fork/Join framework, it still helps with load balancing to create many small tasks
- The forking is O(log n) whereas theoretically other approaches to vector-add is O(1)
- Maps and reductions are the work horses of parallel programming and are by far the two most important common patterns

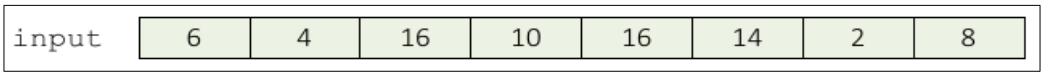
E.3 Parallel Prefix-Sum Algorithm

- This algorithm is like a running total of a one dimensional array of numbers and we want to calculate the total at a particular point
- We can perform an **inclusive** or an **exclusive sum** which speaks on whether we add the value at that point or not
- The DAG arising from this algorithm is linear since we require the previous value and therefore is not highly parallelisable
- The work and the span of this algorithm is the same since they are both equal to the number of node in the tree
- Recall that the divide-and-conquer algorithm has a work of n and a span of log$n$

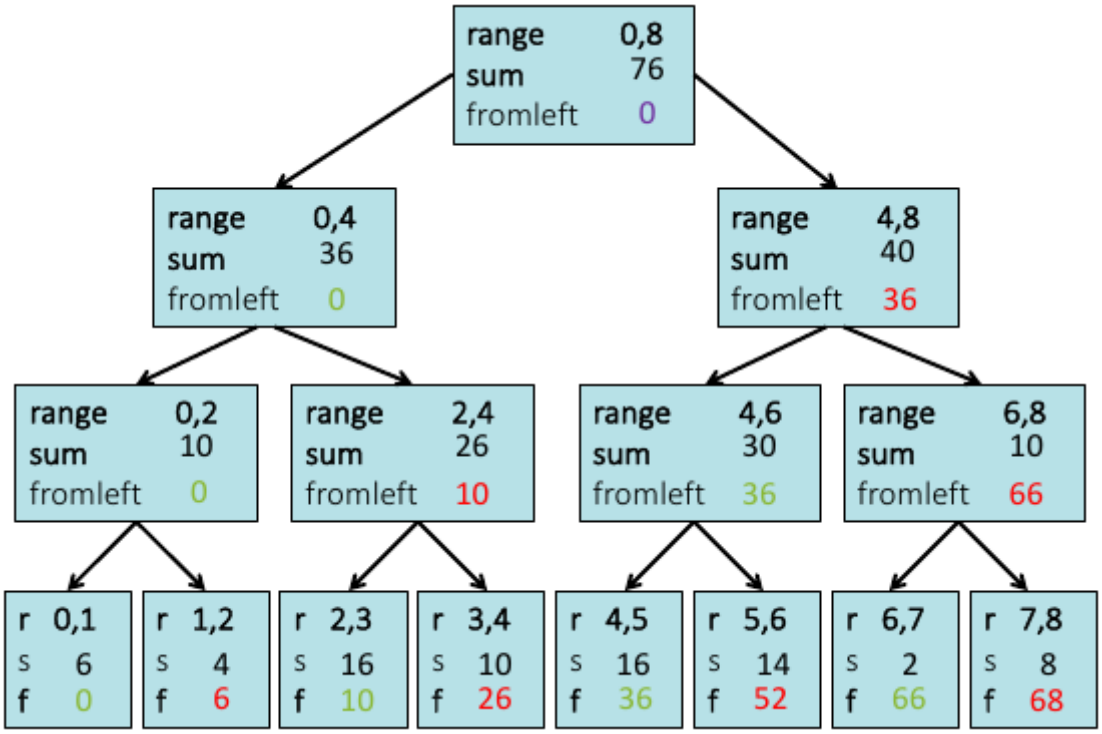
- However, the parallel prefix-sum algorithm does two passes where each one has work of $O(n)$ and a span of $O(\log n)$
- So in total, there is $O(n)$ work and $O(\log n)$ span
- Just like with array summing, the parallelism is $n/\log n$, i.e. work over span with exponential speed-up
- The first pass builds a tree, from bottom-up
- The second pass traverses the tree top-down

Example 3.1: Parallel Prefix-Sum Creating Two Passes

At the beginning of the parallel prefix-sum algorithm, starting with a Fork/Join framework, we begin by counting the number of elements of the entire array. The range of the framework will go from 0 to 8 since the input array has 8 elements.



We then split the array into two and continuing splitting the subsequent arrays recursively until we have a set of arrays with an interval of 1.



The threads in the ForkJoinPool calculate the sum for each of these arrays. We then percolate the sum up the tree using joins to combine results. The top most thread has the total sum of the array, i.e. 76.

The algorithm the calculates the sum from the left value of the array. For the first thread, the sum from left is zero since the thread has the entire array. The right thread then gets the sum of the left thread.

Parallel Prefix-Sum Algorithm

i. **Up Pass**

+ Here, the algorithm builds a binary tree where the root has sum of the range [x, y]
+ If a node has sum [lo, hi] and hi>lo, the left child has sum of [low, middle) (including lo but excluding hi)

This is an easy ForkJoin computation: combine results by building a binary tree with all the range-sums. An algorithm that did not look parallelisable is now capable of being parallelised. The tree us built bottom up in parallel.

ii. **Down Pass**

+ This is the point where we pass down the from left value
+ Root is given a *fromLeft* of *0* since it has no sum to the left
+ Node takes its *fromLeft* value and pass its left child the same *fromLeft* and pass its right child its *fromLeft* plus its left child's sum (as stored during the up pass)

As we can see, this is an easy ForkJoin operation where we traverse the tree built in step 1 and produce no result. Leaves are assigned to output. The invariant is the fromLeft variable which is the sum of elements left of the node's range. The total work is O(n) and the span is O(logn).

We can use this algorithm to parallelise the computation of pivots in QuickSort.

- The sequential cut-off can be added to the framework by considering:
  - in the up pass, the leaf nodes hold the sum of a range
  - in the down pass,

```
output[lo] = fromLeft + input[lo];
 for(i=lo+1; i < hi; i++)
    output[i] = output[i-1] + input[i]
```

- The parallel prefix-sum is highly generalisable

F. Concurrency and Mutual Exclusivity

- So far we have looked at parallel program where threads perform the same task on different data in what is known as **data decomposition**
- We looked at the Fork/Join framework where a large task is broken down recursively into smaller tasks that are the same as the large task
- We avoided race conditions where two threads want to access the same data simultaneously by using **fork** and **join** where **fork** allocated some of its memory to a sub-thread in the ForkJoinPool and did not access that memory again until **join** was used on the sub-thread
- In other words, **join** was the only form of synchronisation used
- In **concurrent programming**, threads are assigned independent tasks on the same resources but do not implement the same algorithm in what is called **task decomposition**
- **Concurrency** refers to correctly and efficiently managing access to shared resources from multiple clients
- Concurrency requires a more complex structure for algorithms since the multiple threads performing different tasks with data, e.g. animator threads, threads that update data
- Unlike parallelism, concurrency is not about implementing algorithms faster but focused on doing many tasks at the same time
- Simultaneous tasks may be necessary for responsiveness
- Simultaneous tasks may be necessary for efficient processor utilization
- They may also be necessary for failure isolation which is a convenient structure if you want to interleave multiple tasks and do not want an exception in on to stop the other

F.1 Synchronisation by Mutual Exclusion

- **Synchronisation constraints** are requirements pertaining to the order of events
- Solving data races involves requires solving synchronisation constraints
- A simple example of synchronisation is **signalling** where an **even A must happen before event B**
- This solved with a **join** as seen in the SumArray example where all the threads must be summed together before the main thread can perform the final sum
- Synchronisation signalling, or serialisation, is that the sub-threads have to signal the main thread when they were done
- The short-coming of synchronisation signalling with **join** is that the program must wait for each thread to finish so we cannot perform other tasks during that time so we cannot solve synchronisation constraints with **join**
- Another form of synchronisation is **mutual exclusion** where **event A and event B must not happen at the same time**
- Mutual exclusion is the programmer's responsibility as the compiler is not involved in deciding safe interleaving in the program
- Mutual exclusion within the critical section implies that the threads must not occur at the same time in that section code
- This typically occurs with shared Objects in Java, for example, UpdateBankAccount would be a critical section and we only want one thread to access it at any time
- Multiple threads compete for the same resource but only one thread can access the resource during that time
- We can use synchronisation to avoid incorrect simultaneous access

- If two processes try to enter a critical section, one process must win and the other must **block** (wait)
- When multiple processes wait, this creates a problem in **contention** which slows down the program
- An example where we need to solve a problem of mutual exclusion is when a counter is shared between threads as shown in the sample code below
- We look at the unsafeCounter first which allows multiple threads to access a counter in an unsafe way and then look at the correct way to implement a counter for multiple threads

Sample Code F.1.1: UnsafeCounter

Counter Class which incremented by each thread.

```
package unsafeCounter;

public class Counter {//could use integer
    private long value;
    Counter() {
        value = 0;
    }

    public long get() {
        return value;
    }

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

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

- If you want to use a primitive type instead of the **Count Object** then the code must include a **copy constructor**
- When sending an Object to multiple threads, the program is accessing the same one

CounterUpdateThread Class is what each class does on the Counter Object *sharedCount.* Primitives must be wrapped into Objects before they are shared among threads.

```
package unsafeCounter;

public class CounterUpdateThread extends Thread {

    Counter sharedCount;
    int rep;

    CounterUpdateThread(Counter c, int repeats) {// wrap primitives into Objects if they are shared
        sharedCount = c; //passing by reference which allows it to be shared between thread
        rep = repeats;//how many times the thread is going to add to the counter
    }

    public void run() {
        //increment up to the number of repeats
        for (int i=0; i<rep;i++) {
            sharedCount.incr();
            // yield(); //to encourage interleavings for exposing race conditions
            // yield is a suggestion to the scheduler to release a thread from the CPU
        }
    }
}
```

TestCounterSafety Class has **main** method and creates 100 threads, each incrementing the value of the **sharedCount Counter** Object. We expect the output to give a final value of 100*100. However, there is a race condition on the Counter object (critical section) so there will be cases where the output is incorrect.

```
package unsafeCounter;

public class TestCounterSafety {
    public static void main (String args [] ) throws InterruptedException {

        int noThrds = 100;// number of threads
        int addPerThread = 100;// add to number of threads. Total additions = 100*100
        Counter sharedCount = new Counter();

        CounterUpdateThread [] thrds = new CounterUpdateThread[noThrds];// create thread array

        for (int i=0; i<noThrds;i++) {
            thrds[i] = new CounterUpdateThread(sharedCount, addPerThread);// create threads
        }
        for (int i=0; i<noThrds;i++) {
            thrds[i].start();
        }
        for (int i=0; i<noThrds;i++) {
            thrds[i].join();// to prevent main thread from ending before all the threads are done
        }

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

}
```

- When you uncomment out the **yield method**, the number of unexpected outputs increases
- The **yield** method increases likelihood of interleaving amongst threads
- This is a great method for checking whether the multi-threaded program is running as expected
- This method works well for a single thread but not for concurrent multiple threading
- The short-coming of this algorithm is exposed due to the increment method **incr()** of the Counter Class which may look like one task but is a set of multiple instructions:

```
1 temp = value;//store value in register
2 temp = temp + 1; //increment the value in register
3 return temp;
```

- The race condition occurs when more than one thread accesses a value x at the same it and add 1 to obtain $x + 1$. These threads all write the value $x + 1$ at the same time so the increment is lost. This is why the value is always less than expected
- When sharing an Object, it must be protected to avoid data races
- To solve this type of race condition, we require operation A and B to be **atomic** with respect to each other
- Operation A and B are **atomic** w.r.t. ea ch 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 said to be **indivisible**.
- For atomicity, we use thread-safe **Atomic Variable Classes** in the **java.util.concurrent** library
- This library contains **AtomicInteger**, **AtomicLong**, **AtomicBoolean**, **AtomicReference**
- There are no atomic variable classes for chars and doubles

- All have **getAndSet()** method that atomically sets the value and returns it in one step
- So to fix our Counter program by making it thread-safe, we can replace the **long** counter with **AtomicLong** to ensure that all updates to the counter state are atomic

Sample Code F.1.2: Counter Program using AtomicLong variable class

```
package unsafeCounter;

public class AtomicCounter {//atomic variable to avoid data race conditions
    private AtomicLong value;
    Counter() {
        value = new AtomicLong(0);
    }

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

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

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

```
package unsafeCounter;

public class AtomicCounterUpdateThread extends Thread {

    AtomicCounter sharedCount;
    int rep;

    AtomicCounterUpdateThread(AtomicCounter c, int repeats) {
        rep = repeats;
    }

    public void run() {
        //increment up to the number of repeats
        for (int i=0; i<rep;i++) {
            sharedCount.incr();
            yield();
        }
    }
}
```

```java
package unsafeCounter;

public class TestAtomicCounterSafety {
    public static void main (String args [] ) throws InterruptedException {

        int noThrds = 100;// number of threads
        int addPerThread = 100;// add to number of threads. Total additions = 100*100
        AtomicCounter sharedCount = new AtomicCounter();

        AtomicCounterUpdateThread [] thrds = new AtomicCounterUpdateThread[noThrds];// create thread
        array

        for (int i=0; i<noThrds;i++) {
            thrds[i] = new AtomicCounterUpdateThread(sharedCount, addPerThread);// create threads
        }
        for (int i=0; i<noThrds;i++) {
            thrds[i].start();
        }
        for (int i=0; i<noThrds;i++) {
            thrds[i].join();// to prevent main thread from ending before all the threads are done
        }

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

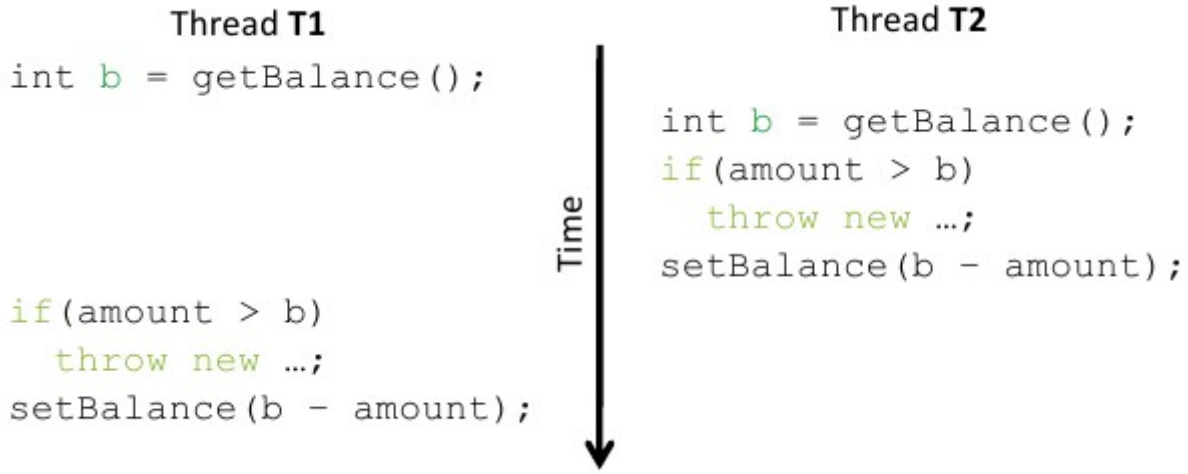
- When a thread A attempts to access an atomic variable while thread B is performing an operation on it, thread A has to wait until B is done. This is a form of mutual exclusion as both threads cannot perform an operation on an atomic variable at the same time
- **Atomic variables only make one class thread-safe if and only if one variable defines the class state and there are no compound actions accessing this state**
- This is achieved using the Counter Class with one AtomicLong variable
- To prevent **state consistency**, one should update Objects related to that state variables in a single atomic operation
- The example in the code shown on the next page illustrate a class which performs a compound action so the atomic variable class cannot be used to make it thread safe
- Although the BankAccount class has one variable which defines the amount of money in the account, we cannot use an atomic variable here because both one condition to apply atomic variable classes to make a class thread-safe has not been met

Sample Code F.1.3: BankAccount Class with compound withdraw action

Using an atomic variable class would only help the **get()** and **set()** methods in the example class below. It would not however, work on the **withdraw()** method because this method is a compound action.

```
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.
}
```

- This would not be a problem for different bank account *x* and *y* when threads call the compound **withdraw** method at the same time, i.e. threads can interleave
- Interleaving of threads in the compound action could cause a lost withdrawal on the same bank account



- Suppose that the initial amount is *b* = 150. Thread T1 accesses *b* first and stores the value as 150
- Suppose that the scheduler releases the CPU from Thread T1 temporarily
- Then Thread T2 performs a **withdraw(100)** which is allowed since the *b* > 100
- When Thread T1 is continued, the value of *b* that is stored is still 150 so it will perform a withdrawal on this amount instead of the value remaining after Thread 2 had completed
- This implies that Thread 1 will also write *b* = 50 and a withdrawal is lost
- We want at most one thread to withdraw from account A at a time
- Java allows us to do this using a **synchronising method** which is a **locking mechanism** where one lock can be held by one thread at a time
- We use **synchronized** on all the methods, including the compound withdraw. This forces the cache to update from main memory

```
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
}
```

- Although it slows some programs down, it is recommended to use **synchronized** on all the methods in the class

- We used **synchronized** on the **getBalance()** and **setBalance()** methods since the class uses a primitive type variable and not an atomic variable. So **synchronized** makes primitives atomic
- Atomic variable classes are faster than using **synchronized**
- Critical sections of code can be made mutually exclusive by prefixing the method with the keyword **synchronized**
- 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
- Note that we can synchronise a method using different syntax as shown:

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

=

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

F.2 Locks for Synchronisation

- When a method is **synchronized** it is **locking** on that Object, which is to say that it holds the **lock** to that Object
- A thread coming up to a **synchronized** method will obtain a lock on the Object
- No other thread can access any of the **synchronized** methods in that class at this same time
- Every Object has a **lock** which controls sections in a class that are protected by this lock
- In other words, if a thread enters a critical section that is locked by Object A another thread can enter a critical section locked by Object B as Objects do not share the same lock
- Objects are locks but not primitive types

- We can use methods to perform operations on a **lock** which is an Abstract Data Type in Java:
  + **new** => generate new lock, initially "*not held*"
  + **acquire** => blocks if this lock is already currently "*held*". Once a lock is released, make lock "*not held*"
  + **release** => makes lock "*not held*"
- We can use locks directly instead of using the **synchronized** keyword as shown in the example code below

Sample Code F.2.1: Using locks directly on BankAccount Object

```
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
}
```

- Locks are primitive mechanisms which, if used incorrectly, may cause the program to hang
- For example, in the sample code above, the **if** condition would cause the program to throw an exception without releasing the lock

- So we would need to release the lock before throwing the exception

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

- If **withdraw** and **deposit** use the same lock, then simultaneous calls to these methods are properly synchronised
- However, **getBalance** and **setBalance** have to acquire the same lock to avoid a race between **setBalance** and **withdraw** that may produce an unexpected result
- To avoid the case where a method is blocked while trying to acquire a lock that it already has, we use **re-entrant locks**
- A **re-entrant lock** remembers the thread that currently holds it and this thread has to release the lock the same number of times that it acquired it
- A **re-entrant lock** has a counter which is 0 when a the lock goes from "*not-held*" to "*held*"
- If the current holder calls **acquire**, the re-entrant lock does not block it, instead, it increments the counter.
- On release, if count > 0, the count is decremented else if count = 0, the lock becomes "*not held*"

G. Thread-Safety

- A class can be considered to be **thread-safe** if it behaves correctly when accessed from multiple threads
- Safe implies that the state of a class is consistent regardless of how the scheduling or interleaving of the execution of those threads
- Safe also requires that there is no additional synchronisation or coordination on the threads that are part of the calling code
- In other words, a class is considered to be thread-safe if no set of operations performed sequentially or concurrently on instances of a thread-safe class
- Writing thread-safe code is about managing an object's state
- A state refers to the instances of the Object. These instances need to be protected from invalid concurrency access
- Race conditions only occur when we have shared, mutable state
  shared:- accessed by multiple threads

  mutable:- value can change
- Whenever more than one thread accesses a given state variable, all accesses must be coordinated using **synchronisation**
- In Java, we can do this using the **synchronized** keyword, volatile variables, explicit locks and/or atomic variables
- A **race condition** is an **error** in a program such that whether the program behave correctly or not depends on the order that threads execute or interleave
- The term race condition can refer to:
  i. **data races**: simultaneous read/write or write/write of the same memory location
  ii. **bad interleaving**: exposing bad intermediate state

- A **data race** is a kind of race condition that occurs when different threads access the same memory location
- At least one of these accesses is a write (read/write, or write/write) and there is no synchronisation that focuses any particular order among these accesses
- Sample code F.1.3 shows an example where we could have had a race condition, that is not a data race, on the **withdraw** method had we removed the **synchronized** keyword
- The general way to protect against data races is to use a **monitor pattern** which refers to Objects that are guarded by a **lock** (monitor's access to the Object)
- The same lock is used whenever the object is accessed
- An Object following the Java monitor pattern will synchronise all the methods in the class
- All mutable states are guarded with the Object's own in intrinsic lock
- This can become an issue if the Object is accessed by many threads as it leads to problems with contention
- Essentially, thread-safety is achieved by serialising all accesses to a collection's state and process only one request at a time

G.1 Compound Actions and Thread Safety

- For thread-safety, it is not enough to declare every method of every shared object **synchronized**
- For example, the **Vector Class** in Java has all its methods declared as **synchronized**
- However, we still obtain a race condition on the following code statement as it is not atomic compound action

```
if (!vector.contains(element))
    vector.add(element);
```

- So we know that no other element can access **contains** and **add** while another thread is using
- We still get a race condition because this is a **check-then-act** because we are checking the state of the class and then changing it
- The race condition in the compound action leads to duplication of elements added to the Vector Object because two threads could possibly get **false** for **contains** for the same element which would lead them to both **add** element to the Vector Object
- To synchronise the whole block, we would need to synchronise on the Vector

synchronized (Vector) {<compound action>**}**

- This would synchronise on the class's intrinsic lock
- So we have to synchronise on the compound action and the class
- Another type of compound action is a **read-modify-write** compound action where the resulting state is derived from the previous state

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    temp = temp + 1;
    return temp;
  }
}
```

- Remember that incrementing is a type of read-modify-write compound action
- Finding the maximum value of an array or a series of numbers is a check-then-act compound action where each thread checks a part of the series

- Below is another check-then-act compound action which uses some expensive Object (meaning it takes up a lot of memory)
- Ideally, we do not want to create this expensive Object until we need it
- As soon as a thread needs it, it checks whether an instance of the Object already exists and if it does not, an instance is create

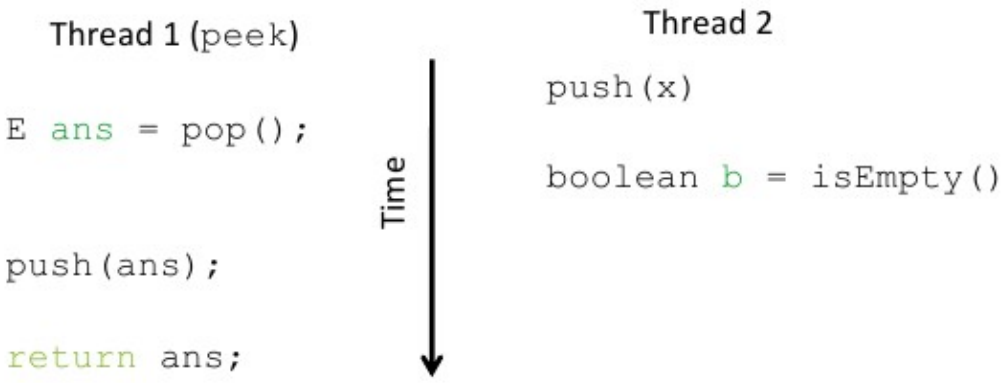
```
@NotThreadSafe
public class LazyInitRace {

  private expensiveObject instance = null;
  public ExpensiveObject getInstance() {
   if (instance==null)
      instance = new ExpensiveObject();
   return instance;
}
}
```

- This is not thread-safe as multiple expensive Objects could be created at the same time after more than one threads reaches the check statement and sees that the Object does not exist
- An example where a compound action may cause **bad interleaving** is shown in the code on the next page where a **Stack** class uses a combination of synchronised and non-synchronised methods in an attempt to achieve thread-safety
- All the methods of the class are synchronised except for the **peek** method which implements the synchronised **pop** and **push** methods
- The method **peek** attempts to avoid a race condition on the value in the stack by popping it out, checking it and then replacing it into the stack
- However, this is incorrect as it may lead to bad interleaving when one thread pops the value from the stack and another thread comes in to check for the value before it is pushed back into the stack by the thread that was reading it

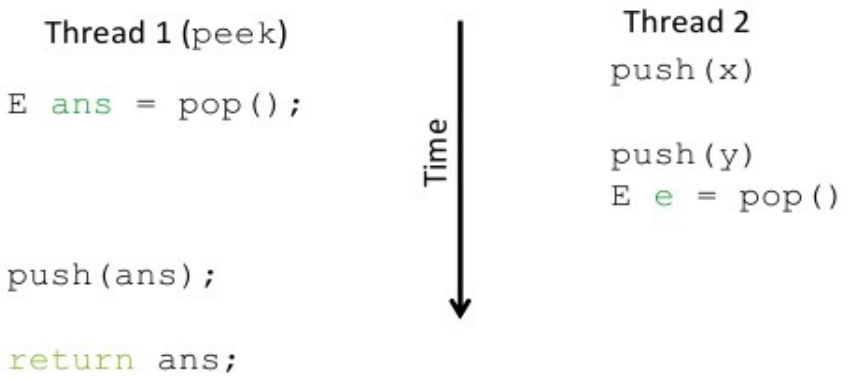
- This bad interleaving will cause unexpected results

```
class Stack<E> {
  … // state used by isEmpty, push, pop
  synchronized boolean isEmpty() { … }
  synchronized void push(E val) { … }
  synchronized E pop() {
    if(isEmpty())
      throw new StackEmptyException();
    …
  }
  E peek() { // this is wrong
     E ans = pop(); //take off list and copy
     push(ans); // put back on list
     return ans;
  }
}
```

- When thread 2 uses the synchronised **isEmpty** method after thread 1 has used **pop**, **isEmpty** will return **true** since the value has not been pushed back into the stack by thread 1

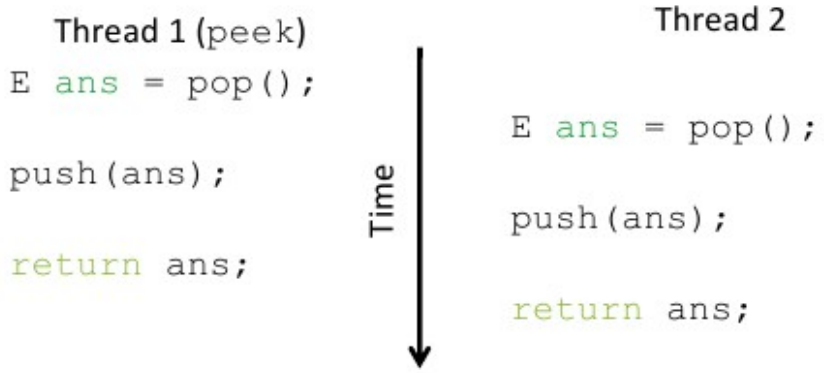


- Another bad interleaving occurs when values are pushed back into the stack after being popped out in a LIFO ordering



- The order will change (i.e. should get *x* then *y* but obtain *y* then *x*) after the threads execute in this order

- This is because thread 1 pops the value *x* out of the stack followed by thread 2 pushing the value *y* in the stack before thread 1 pushes the popped value *x* back into the stack
- This violates the LIFO property that we intended to maintain
- Another bad interleaving is shown below where we accidentally pop an item that should be in the stack but is temporarily not there
- Then the method **peek** will throw an exception as the number of pushes will be less than the number of pops



- To fix this and prevent cases of **bad interleaving**, we need to synchronise **peek** with the critical section so that a re-entrant lock can allow calls to **push** and **pop**
- We can do this by synchronising **peek** within the stack which is recommended as it follows the Java monitor pattern or we can synchronise on the stack in an external case as shown

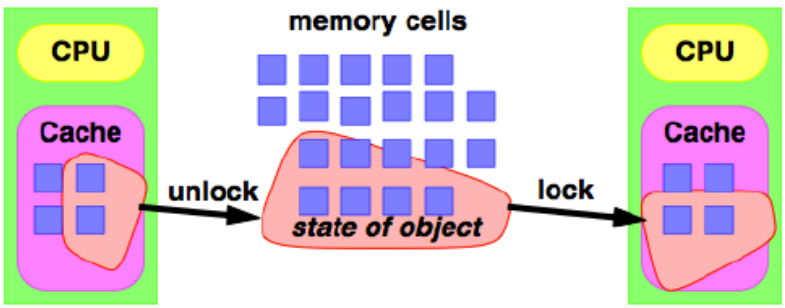
```
class Stack<E> {
  …
  synchronized E peek(){
     E ans = pop();
     push(ans);
     return ans;
  }
}
```

```
class C {
  <E> E myPeek(Stack<E> s){
    synchronized (s) {
      E ans = s.pop();
      s.push(ans);
      return ans;
    }
  }
}
```

- These methods however do not fix the issue about visibility

G.2 Thread-Safety and Visibility of Objects

- Synchronisation is not only about atomicity but is also about **memory visibility** (also known as **cache coherence**)
- When a thread modifies an Object, we need to ensure that other threads can **see the changes** that were made
- Without synchronisation, this may never happen
- Every CPU has one or multiple tiers of fast and temporary memory called the **cache** where variables are written to temporarily during processes
- There is no guarantee that variables in the cache will ever be written to main memory and this may lead to unsafe execution and unexpected outputs
- Unless synchronisation is used every time a shared variable is accessed, it is possible to see a **stale value** for that variable
- We call it a stale variable because it has been updated in one cache while another cache holds the value in the previous state of the class
- We do have cases where a state becomes nonsensical when some parts of an Object are more stale than others
- To avoid stale values, synchronise all the getters and setters of Object instance variables on the same lock
- Locking generates messages between threads and memory-hierarchy
- Lock acquisition forces reads from memory to thread cache
- Lock release forces writes of cached updates to memory



- A way to get around this is to use **volatile** keyword which control per-variable cache flush or reload
- It is not recommended to use volatile to achieve thread-safety
- When a field is declared volatile, it is not cached
- A read action on a volatile variable always returns the most recent write in all the caches
- This method is a lighter weight mechanism than **synchronized** because it does not use locking
- Since it does not use locking, accessing a volatile variable cannot cause another thread to block
- Volatile variables are faster than most locking mechanisms but is generally slower than regular variables
- Volatile has limited usability as it does not protect compound actions
- The most common use of **volatile** is for a **flag** variable

```
volatile boolean asleep;
while (!asleep)
 countSomeSheep();
```

- While locking can guarantee both visibility and atomicity, volatile variables can only guarantee visibility
- Threads do not share primitive types because Java is pass-by-value for primitives so it is better to use AtomicBoolean for boolean flags

```
static AtomicBoolean started;
static AtomicBoolean pause;
static AtomicBoolean done;
static AtomicBoolean won;
```

- Threads check the flags when running to see the state of the program
- This is a simple solution for signalling but is not perfect for all situations

- If for example a thread sees a *pause* flag, it enters what is known as the spin-state where it continuously checks if the flag is still up
- We make flags static because they belong to the main class
- If we used a normal boolean, then Java would use pass-by-value and make copies of the flag so that when it is changed by one thread, the change cannot be seen by other running threads
- Using atomic variables ensures that the state of the class is always updated in caches
- The compiler/hardware will never perform a memory reordering that affects the result of a single-threaded program
- The compiler/hardware will never perform a memory reordering that affects the result of a data-race multi-threaded program
- If no interleaving of the program has a data race, then we can neglect reordering
- Naturally, we can use synchronization to avoid data races
- The following sample code illustrates a data race on variables *x* and *y* which causes an assertion failure

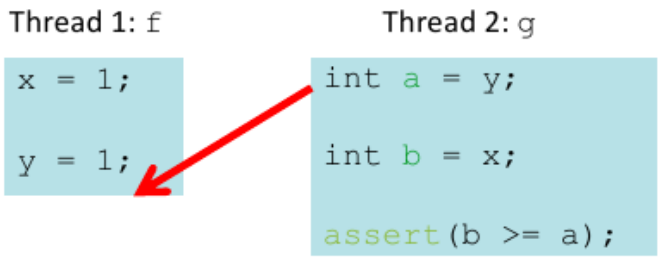
Sample Code G.2.1: Data race on a seemingly benign code

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

  void f() {
    x = 1;
    y = 1;
  }
  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

- At first glance, there is no interleaving of **f** and **g** where the assertion fails

- We can also prove that there is no case where **f** and **g** interleave such that *b* < *a* which causes the program to fail
- *Proof by Contradiction*:

  If *b* < *a*, then *a* == 1 and *b* == 0. If *a* == 1, then *a* = *y* happened after *y* == 1. And since programs execute in order, *b* == *x* happened after *a* = *y* and *x* = 1 happened before *y* = 1. So by transitivity, *b* == 1 which is a contradiction.
- The problem arises because the compiler/hardware does not guarantee that a thread will perform operations in order as long as the reordering is not detectable
- When the compiler runs **f** in thread 1, there is no dependency on the ordering of the assignment of 1 to *x* and *y*, i.e. the compiler can sometimes start with *y* = 1 even though the intended ordering is different => leads to data race on *x*, *y*



- The reordering of the operations perform in thread one is apparent to thread 2 which may start reading the value of *x* or *y* for *a* and *b* while thread 1 is still running
- This data race is due to the unsynchronised read/write or write/write of same location
- The interleaving of the threads by the compiler does not affect the logic of a program that is data race free

Sample Code G.2.2: Fixing the data race in the benign code

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

- The Java compiler does not reorder synchronised blocks
- The **Java Memory model** defines how threads interact through memory and contains rules for which values may be seen by a read of shared memory that is updated by multiple threads
- As the specification is similar to the memory models for different hardware architectures, these semantics are known as the Java programming language memory model
- Java memory model requires maintenance of *within thread as-if-serial* semantics
- Each thread must have same result as if executed in serial
- The JVM defines a partial reordering called *happens-before* on all actions in a program
- To guarantee that an action B sees the results of action A, there must be a *happens-before* relationship between them
- The Java Memory model is specified in *happens-before* rules
- The **monitor lock rule** of the Java Memory model guarantees that a release of a lock happens before every subsequent acquire of the same lock
- The **volatile variable rule** states that a write of a volatile variable happens before every subsequent read of the same volatile variable

- **Out-of-thin-air safety** guarantees that when a thread reads a variable without synchronisation, it may see a stale value but it will be a value that was actually written at some point
- This does not apply for 64-bit numeric values when running a 32-bit architecture
- The JVM can read or write these in 2 separate 32-bit operations
- Shared mutable **double** and **long** values must be declared volatile or guarded by a lock when using a 32-bit CPU

G.3 Java Synchronizers

- A **Java synchronizer** is any object that coordinates the control flow of threads, e.g. **join**, **wait**
- Java also has many others, including:
  + Latches
  + Barriers
  + Blocking Queues

G.3.1 Latches

- Latches act as a gate where no thread can pass until the gate opens, and then all threads can pass
- Latches delay progress of threads until it enters a terminal state
- Latches cannot then change state again as it opens forever
- The **CountDownLatch** is the main latch in the **java.util.concurrent** library which allows one or more threads to wait until a set of operations being performed in other threads completes

```
CountDownLatch = new CountDownLatch(3); //countdown from 3
```

- It is used to ensure that threads wait to perform a task
- The program calls **await** on the latch at the point where we require that a thread waits until the value of the latch becomes 0
- A thread needs to call **countDown** so that when another thread is done with its task, it can decrement the count of the latch, releasing all waiting threads when the count down reaches 0
- The example below uses two CountDownLatch latches called startSignal and doneSignal
- We could use a CountDownLatch instead of **join** to ensure that threads wait until the value reaches zero

Sample Code G.3.1: CountDownLatch implementation

```
class Driver { // ...
  void main() throws InterruptedException {
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(N);

    for (int i = 0; i < N; ++i) // create and start threads
      new Thread(new Worker(startSignal, doneSignal)).start();

    doSomethingElse();          // don't let run yet
    startSignal.countDown();    // let all threads proceed
    doSomethingElse();
    doneSignal.await();         // wait for all to finish
  }
}

class Worker implements Runnable {
  private final CountDownLatch startSignal;
  private final CountDownLatch doneSignal;
  Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
    this.startSignal = startSignal;
    this.doneSignal = doneSignal;
  }
  public void run() {
    try {
      startSignal.await();
      doWork();
      doneSignal.countDown();
    } catch (InterruptedException ex) {} // return;
  }
  void doWork() { ... }
}
```

- When a Worker thread is created is created, it is sent a latch and because these latches are Objects, they are passed-by-reference
- The threads are therefore effectively sharing a latch
- The main thread signals the threads to start and then it waits until all the threads are done
- Each sub-thread is going to countdown when it is finished by giving a doneSignal

G.3.2 Barriers

- Barriers are similar to latches in that they block a group of threads until an event has occurred, however, barriers wait for other threads while latches wait for events
- **CyclicBarrier** allows a fixed number of parties to rendezvous repeatedly at a barrier point
- Once the CyclicBarrier has opened for a certain specified number of threads, it closes again
- Once all threads are there, the barrier is passed, all threads are released and the barrier is reset
- CyclicBarrier has an optional *barrier* action Runnable command
- These are particularly useful when coding for simulators that work with time-steps to indicate that the program cannot progress to the next time-step until all the threads have reached a certain point
- The optional barrier action will run once per barrier point, after the last thread in the party arrives but before any threads are released
- Barrier action is useful for updating shared-state before any of the parties continue
- The next example illustrates CyclicBarrier and also how coarse synchronisation can slow down concurrent code
- The **DisplayImage** class is the main class which creates a window and takes in some arguments on how big the window should be
- This class uses a CountDownLatch instead of **join** so that the latch is released when all the threads are done

Sample Code G.3.2: ThreadRaces illustrating CyclicBarrier

```
import java.awt.FlowLayout;
import java.awt.image.BufferedImage;
import java.io.*;
import java.util.Arrays;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.CyclicBarrier;

import javax.imageio.*;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

//do filter of image in serial
class DisplayImage{
    public static void main(String[] args) throws InterruptedException, IOException {
    »
        int w = 500; // window width
        int h=500; //window height
    »
    »   //deal with command line arguments if provided
    »   if (args.length==2)  {
    »   »   w=Integer.parseInt(args[0]); // width provided
    »   »   h=Integer.parseInt(args[1]); // width provided» »   »
    »   }
    »   System.out.println("Using image  of size "+w + "x"+h );
        BufferedImage dish =
                new BufferedImage(w, h, BufferedImage.TYPE_INT_ARGB);
        ImageIcon icon=new ImageIcon(dish);
        JFrame frame=new JFrame();
        frame.setLayout(new FlowLayout());
        frame.setSize(w,h);
        JLabel lbl=new JLabel();
        lbl.setIcon(icon);
        frame.add(lbl);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        int noThreads=6;
        Thread []cols= new Thread[noThreads];

        CountDownLatch allDone = new CountDownLatch(noThreads);
        CyclicBarrier middle_barrier= new CyclicBarrier(noThreads);

       // for(int i=0,y=0;i<noThreads;i++,y+=20) {
        cols[0] =new Thread( new HThread(10,490,100,150,dish,
        »   »   new Color(255,0,0),allDone, middle_barrier));
        cols[1] =new Thread( new HThread(10,490,200,250,dish,
        »   »   new Color(255,0,0),allDone, middle_barrier));
        cols[2] =new Thread( new VThread(200,250,10,490,dish,
        »   »   new Color(0,255,0),allDone, middle_barrier));
        cols[3] =new Thread( new VThread(100,150,10,490,dish,
        »   »   new Color(0,255,0),allDone, middle_barrier));
        cols[4] =new Thread( new HThread(10,490,300,350,dish,
        »   »   new Color(255,0,0),allDone, middle_barrier));
        cols[5] =new Thread( new VThread(300,350,10,490,dish,
        »   »   new Color(0,255,0),allDone, middle_barrier));


        »
       // }
        for(int i=0;i<noThreads;i++) {cols[i].start();}
        Thread frame_painter = new Thread () { public void run() { while (true) frame.repaint(); }};
        frame_painter.start();
        allDone.await(); //instead of lots of joins
        File dstFile = new File("output.png");
        ImageIO.write(dish, "png", dstFile);


    }
}
```

- We want all the threads to wait for the *middle_barrier* when they reach the middle of the image
- The vertical and horizontal thread, **VThread** and **HThread**, classes have a shared CountDownLatch and a CyclicBarrier

G.3.3 Lock Granularity

- Lock granularity is used to make design decisions about the size of the synchronised block and the parts of the Object that should be locked
- The **coarse-grained** lock structure uses fewer locks, i.e. more objects per lock, e.g. one lock for entire data structure, e.g. array
- The **fine-grained** structure uses more locks, i.e. fewer objects per lock, e.g. one lock per data element like an array index or one lock per bank account
- **Coarse-grained:**
  + simpler to implement
  + faster to implement operations that access multiple locations because they're all guarded by the same lock
  + easier operations that modify data-structure shape
  + increases the risk of liveness and performance problems
- Liveness here refers to the starvation of processes or slowness of the progression of the problem
- **Fine-grained**:
  + more simultaneous access (performance increase when coarse-grained would lead to unnecessary blocking)
  + increases risk of exposing data races
  + increases chances of liveness problems and run risks of deadlocks

- The strategy is to start with the coarse-grained lock design and move to the fine-grained design for better performance only if contention on the coarser locks becomes an issue and can avoid race conditions

Example G.3.3: Coarse-grained vs Fine-grained Lock Structures

HashTables

- We can use the coarse-grained implementation which uses one lock for the entire HashTable or we can implement the fine-grained structure where each bucket has a lock
- Consider that fine-grained supports more concurrency for **insert** and **lookup** as it allows more processes to run at the same time
- Resizing a HashTable consists of choosing a new hash function to map to the new size, creating a HashTable of the new size, iterating through the elements of the old table, and inserting them into the new table
- Consider that implementing **resize** is easier with coarse-grained because we can simply lock the whole HashTable
- If a HashTable has a *numElements* field, maintaining it will destroy the benefits of using separate locks for each bucket

G.3.4 Critical-section Granularity

- A related granularity issue is critical-section size which deals with how much work to do while holding a lock or many locks
- If critical-sections run for too long, a program may lose performance because other threads are blocked
- If critical sections are too short the program ends up in an unsafe state and may get more race conditions
- The point is not to make critical-sections too expensive and to be aware of an increases chance of race conditions

Example G.3.4: Critical-section granularity with expensive object

Suppose that we want to change the value for a key in a HashTable without removing it from the table. Below, we lock the whole table which has an expensive object. Using this method slows down threads that want to access the object which makes the program too slow:

```
synchronized(lock) {
  v1 = table.lookup(k);
  v2 = expensive(v1);
  table.remove(k);
  table.insert(k,v2);
}
```

Suppose that we decide to separately lock on the **lookup**, do the expensive operation and then use the same lock to do the **insert**. This exposes a read-modify-write compound action.

```
synchronized(lock) {
  v1 = table.lookup(k);
}
v2 = expensive(v1);
synchronized(lock) {
  table.remove(k);
  table.insert(k,v2);
}
```

The critical-section granularity here is just right. This section of code does a **lookup** and then separately, in another critical-section, reads again before performing the **insert**. This is done in a **while** loop.

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

H. Thread-Liveness

- **Concurrent correctness** refers to **safety properties** and **liveness property**
- For safety, the correctness property must always be true
- For liveness, the property must eventually become true and processes must make progress and not wait indefinitely
- **Starvation** refers to indefinite blocking
- A starvation-free state is one where every thread trying to acquire the lock eventually succeeds
- **Deadlock** is when two or more processes are waiting indefinitely for an event that can never occur because the event is waiting for a process that is itself waiting
- A deadlock-free concurrent program is one where some thread trying to acquire a lock eventually succeeds
- We use locking to ensure safety but locks are inherently vulnerable to deadlock
- By locking indiscriminately, we can cause **lock-ordering deadlocks** as shown in the example below

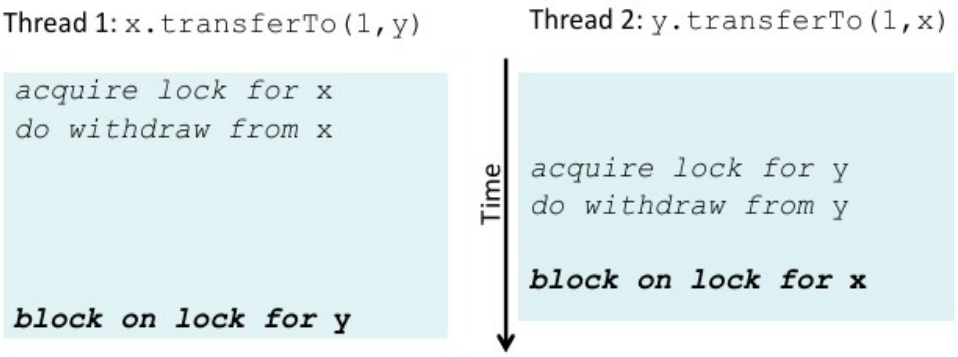
Example H.1.1: BankAccount deadlock

Suppose that we have a BankAccount class with the addition of a transfer method which withdraws an amount from **this** bank account and deposits it into another bank account *a*.

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

This withdrawal and deposit procedure may lead to deadlock. Deadlocks occur when there is more than one lock involved. Waiting for one lock is a starvation issue and not a deadlock issue.

During a call to *a.deposit*, thread holds 2 locks at the same time. To understand when we can get a deadlock, suppose that thread 1 transfers an amount of 1 from bank account *x* to bank account *y* and thread 2 transfers the same amount from *y* to *x*.



Thread 1 acquires the lock on *x,* withdraws from *x* and releases the CPU. Then thread 2 comes in and acquires the lock on *y* to withdraw from it. When thread 2 attempts to acquire the lock on *x* to do a deposit, thread 2 is blocked as thread 1 still has the lock on *x*. The OS scheduler attempts to continue with thread 1 by acquiring the lock on *y* for the deposit but is block since thread 1 still has the lock on it. This results in a deadlock known as the **deadly embrace**.

H.1.1 Deadlocks in Java

- Simplest form of deadlock is the **deadly embrace** where thread A holds lock L while trying to acquire lock M, and thread B holds lock M while trying to acquire lock L
- An illustration of this scenario is shown in example H.1.1 above
- The BankAccount class can be made deadlock-free by synchronising on the parts that will cause deadlock

```
class BankAccount {
  …
  private int acctNumber; // must be unique
  void transferTo(int amt, BankAccount a) {
    if(this.acctNumber < a.acctNumber)
       synchronized(this) {
       synchronized(a) {
          this.withdraw(amt);
          a.deposit(amt);
       }}
    else
       synchronized(a) {
       synchronized(this) {
          this.withdraw(amt);
          a.deposit(amt);
       }}
  }
}
```


- **Lock-ordering deadlocks** occur when two threads attempt to acquire the same locks in different order
- A program will be free of lock-ordering deadlocks if all threads acquire the locks they need in a fixed global order
- This requires global analysis of programs locking behaviour
- A program that never acquires more than one lock at a time will also never deadlock but is often impractical

- Another example is the one shown below which uses the thread-safe StringBuffer class

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

- We have the potential for the StringBuffers to deadlock when two buffers try to append to each other
- We can also get a **lock-ordering deadlock** because the two buffers can get the locks in reverse order
- We also have a compound action which is not synchronised between getting the length and getting the characters