Parallel Programming Summary

Gregory Rozanski

April 16, 2020

0.1 Mutual Exclusion

0.1.1 Definitions

Concurrence:
A form of computing in which several computations are executed during overlapping time periods i.e "concurrently" instead of "sequentially". A concurrent system is one where a computation can advance without waiting for all other computations to complete.

Concurrency Control:
Concurrency control ensures that correct results for concurrent operations are generated while getting those results as quickly as possible.

Race condition:
A race condition or race hazard is the condition of an electronics,software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events.

Scheduling:
Scheduling is the method by which work is assigned to resources that complete the work. Schedulers are often implemented so they keep all computer resources busy, allow multiple users to share system resources effectively, etc.

Thread of execution:
A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler.

Critical section:
Concurrent accesses to shared resources can lead to unexpected or erroneous behaviour so parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access. This protected section is the critical section/region. It cannot be executed by more than one process at a time.

Deadlock:
Problem Description:
The problem which mutual exclusion addresses is a problem of resource sharing: how can a software system control multiple processes access to a shared resource, when each process needs exclusive control of that resource while doing its work?
Solution:
The mutual-exclusion solution to this makes the shared resource available only while the process is in the critical section. It controls access to the shared resource by controlling each mutual execution of that part of its program where the resource would be used.

Language features vs. parallelism: Guidelines

- Keep variables as local as possible: global variables can be accessed by various parallel activities
- If possible, avoid aliasing of references: aliasing can lead to unexpected updates to memory through a process that accesses a seemingly unrelated variable (named differently)
- If possible avoid mutable state, in particular when aliased: aliasing is no problem if the shared object is immutable

Multitasking Concurrent execution of multiple tasks/processes. If you only have one CPU its called multiplexing. If we switch fast enough between the processes multiplexing gives us the feeling that the processes are running in parallel. This is advantegous because the CPU usually waits for inputs and outputs of the memory, hence we can run other processes during this waiting period to increase efficiency.

Process context A process is a program executing inside an Operating System. Each running instance of a program is a seperate process. Each process has a context:

- Instruction counter: points to next instruction
- Values in registers, stack and heap
- Resource handles
- ...

When switching between context we have to temporarily save the context of the current process and load the context of the next process

process lifecycle

1. Process created (load from disk)
2. Process in waiting state (Pool of processes which can be executed)
3. Scheduler picks the process and puts it in a Running state
4. Process can enter Blocked state (usually because of I/O, hence it cannot be executed). When block is released it returns to a waiting state
5. Process enters Terminated state, where context is deleted

Each process demands a certain amount of the main memory. In the case where there isnt enough left, swapping takes place in which the context of the current process is put on the Hard Disk creating space for another process. (Slows down the system).

Context Switch When switching between two processes, the OS interrupts the first one captures its state, loads the state of the second process and executes it. There is alot of overhead generated when switching between processes, hence switching alot between processes is inefficient.

Threads Threads are:

- independent sequences of execution
- running in the same OS process

Multiple threads share the same address space hence they execute different code but share the same memory. Threads have the advantage that they are not controlled by protocols and can read and write freely (this also makes it more vulnerable to programming mistakes) .

- Threads are not shielded from each other
- Threads share resources and can communicate more easily

Context switching between threads is efficient

- no change of address space
- no automatic scheduling
- no saving/reloading of PCB (OS process) state

Multithreading 1 vs. many CPU's When multiple threads share a single CPU then the threads take turns executing and the others are put in a waiting state. With multiple CPU's e.g 3 threads, 3 CPUs all threads can run constantly increasing performance.

Java Threads JVM implementation of the thread concepts i.e parallel execution. It is a set of instructions to be executed one at a time, in a specific order. Thread class is part of the core language. Every Java program has at least one execution thread (first one calls main()). A Program ends when all threads finish. Threads can continue to run even if main() returns. Creating a Thread object or calling run() does not start a thread we need to call start().

java.lang.Thread

- start() : method called to spawn a new thread (causes JVM to call run() method on object)
- interrupt() : freeze and throw exception to thread (used to terminate a thread at time of call)
- sleep(int num) : puts thread to sleep for num ms
- getID(): gets the thread's ID
- getName() : gets the name of the currentThread
- setName() : sets the name of the currentThread
- currentThread() : returns current Thread
- setPriority(int num) : Threads can have a priority between 1 and 10. JVM uses the priority of threads to select the one that uses the CPU at each moment. The Scheduler decides wether or not to regard the priority of the threads.
- getState() : Denotes the status the thread is in
- join(): thread finishes and returns the result to the sleeping main thread (May throw InterruptedException)
- wait() : Consumer goes to sleep i.e status NOT RUNNABLE. If the thread has the lock and is in a state where it cant do anything productive, wait is called such that other threads can access the resource (can only be used if the thread holds the lock). It is recommended to use a while loop around the condition, in order to see that the thread returned from the wait() at a valid time. When not specifying myObject.wait() then wait() = this.wait().
- notify()/notifyAll() : Changes the state of all threads waiting on the resource to Runnable (can only be used if the thread holds the lock i.e in synchronized block). notify() wakes the highest-priority thread closest to front of object's internal queue. When not specifying myObject.notify() then notify() = this.notify()

0.2 Creating Java Threads

OPTION 1: Instantiate a subclass of java.lang.Thread class

- Override run method
- run() is called when execution of that thread begins
- A thread terminates when run() returns
- start() method invokes run()
- calling run() does not create a new thread!

```
class ConcurrWriter extends Thread {
       public void run() {
               // code here executes concurrently with caller;
       }
}
ConcurrWriter writerThread = new ConcurrWriter();
writerThread.start(); // calls ConcurrWriter.run()
```

OPTION 2: Use Runnable Interface

- single method: public void run()
- class implements Runnable

```
public class ConcurrWriter implements Runnable {
        public void run() {
                // code here executes concurrently with caller;
        }
}
ConcurrReader readerThread = new ConcurrReader();
Thread t = new Thread(readerThread);
t.start(); // calls ConcurrWriter.run()
```

Here there it is distinguished between how the programm is executed (the thread) and what is being executed (Runnable)

Busy Waiting: By spinning(looping) until each worker's state is TERMINATED. Join (sleep, wakeup) typically incurs context switch overhead. If worker threads are short-lived, busy waiting may perform better.

Exceptions: Exceptions in a single threaded (sequential) program terminate the program, if not caught. If a worker thread throws an exception, the exception is shown on the console, the behaviour of thread.join() is unaffected, hence the main thread may not be aware of an exception inside a worker thread. Implementing UncaughtExceptionHandler interface allows us to handle unchecked exceptions. Three options:

- Register exception handler with Thread object
- Register exception handler with ThreadGroup object
- Use setDefaultUncaughtExceptionHandler() to register handler for all threads Handler can then record which threads terminated exceptionally or restart them, or ...

```
public class ExceptionHandler implements UncaughtExceptionHandler {
        public public Set<Thread> threads = new HashSet<>();{


@Override
public void uncaughtException(Thread thread, Throwable throwable){
        println("An exception has been captured");
        println(thread.getName());
        println(throwable.getMessage());
        . . .
        threads.add(thread);
        }
}


public class Main {
        public static void main(String[] args) {
        . . .
        ExceptionHandler handler = new ExceptionHandler();
        thread.setUncaughtExceptionHandler(handler);
        . . .
        thread.join();
        if (handler.threads.contains(thread)){
           //bad
        } else {
           // good
        }
   }
}
```

Thread Safety: This implies program safety and refers to "nothing bad ever happens", in any possible interleaving.

Liveness: "eventually something good happens" (e.g endless loops are an example of liveness hazards in sequential programming). Threads makes liveness hazards more frequent: If ThreadA holds a resource(e.g a file handle) exclusively, then ThreadB might be waiting for that resource forever. Hence liveness means that progress will be made.

Examples of safety properties:

- absence of data races
- mutual exclusion
- linearizability
- atomicity
- schedule-deterministic
- absence of deadlock
- custom invariants

Synchronized Multiple threads may read/write the same data (shared objects,global data). To avoid bad interleaving we use explicit synchronization. In Java, all objects have an internal lock, called intrinsic/monitor lock. Synchronized operations lock the object, hence no other thread can successfully lock and use the object and must wait until the lock is freed. Generally if accessing shared memory, make sure itis done under a lock, if not the code is prone to a data race.

Synchronized Methods: A synchronized method grabs the object or class's lock at the start , runs to completion, then releases the lock. This is useful for methods whose entire bodies are critical sections, and thus should not be entered by multiple threads at the same time. A synchronized method is a critical section with guaranteed mutual exclusion

```
// synchronized method: locks on "this" object
public synchronized type name(parameters) {...}

// synchronized static method: locks on the given class
public static synchronized type name(parameters) {...}


Synchronized Blocks:

synchronized (object) {
    statement(s); //critical sections
}
```

Synchronized Blocks: Enforces mutual exclusion with regards to some object. Every Java object can act as a lock for concurrency: A thread T_1 can ask to run a block of code, synchronized on a given object O. The synchronized block makes sure there is no interleavings of the statements inside the block, it does not prevent other threads from executing statements outside of the block, hence it is still possible for bad interleavings to happen with statements outside the block

- If no other thread has locked O, then T_1 locks the object and proceeds
- If another thread T_2 has already locked O, then T_1 becomes blocked and must wait until T_2 is finished with O (that is, unlocks O). Then, T_1 is woken up, and can proceed

Reentrant: Locks are recursive. A thread can request to lock an object it has already locked, and will lock it, the thread will then release the lock multiple times.

Synchronization granularity: Using multiple locks to allow multiple threads to work on code while still being protected.

Synchronized and Exception If an exception is triggered in the middle of a synchronized bock, then the lock released, as if the synchronized scope ends right at the point where the exception is thrown. When the exception is caught, then the exception handler is executed. If there is no exception handler, then the exception is propagated back down to the caller of the method. Any side effects are not reverted, they do take effect even if exceptions are thrown.

Producer-Consumer: The Producer puts items into a shared buffer (shared resource), the consumer takes them out, consumption is only possible if buffer isn't empty.

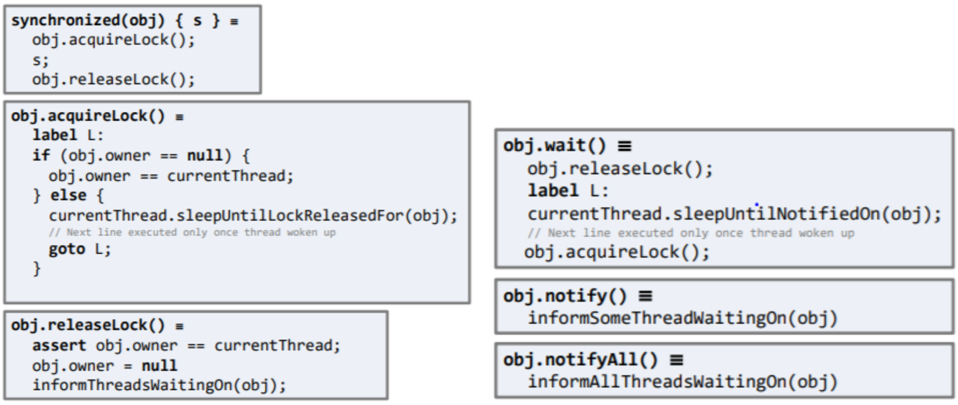
Pseudo-Code Implementation of synchronized block :

0.3 Parallel Architectures

Parallelism: Use extra resources to solve a problem faster

Concurrency: Correctly and efficiently manage access to shared resources.

Distributed computing: Physical separation, administrative separation, different domains, multiple systems



```
synchronized(obj) { s } ≡
  obj.acquireLock();
  s;
  obj.releaseLock();
```

```
obj.acquireLock() ≡
  label L:
  if (obj.owner == null) {
    obj.owner == currentThread;
  } else {
    currentThread.sleepUntilLockReleasedFor(obj);
    // Next line executed only once thread woken up
    goto L;
  }
```

```
obj.releaseLock() ≡
  assert obj.owner == currentThread;
  obj.owner = null
  informThreadsWaitingOn(obj);
```

```
obj.wait() ≡
  obj.releaseLock();
  label L:
  currentThread.sleepUntilNotifiedOn(obj);
  // Next line executed only once thread woken up
 obj.acquireLock();
```

```
obj.notify() ≡
  informSomeThreadWaitingOn(obj)
```

```
obj.notifyAll() ≡
  informAllThreadsWaitingOn(obj)
```

Figure 1: Pseudo implementation of synchronized block

Von Neumann architecture: Program data and program instructions share the same memory.

CPUs and Memory Hierarchies Caches are preloaded data readily available to speed up access time.

- Goal: Allow cores to work int parallel, on their own, fast memory
- CPU reads/writes values from/to main memory, to compute with them, with a hierarchy of memory caches in between. Faster memory is more expensive, hence smaller: L1 is 5x faster than L2, which is 30x faster than main memory, which is 350x faster than disk.
- Synchronisation between caches is taken care of by cache coherence protocols(e.g MESI see notes page 3)
- Concurrency Hazard: cores may pre-/postpone reads/writes from/to cache; memory barriers needed to prevent problems with parallel code. (In Java memory barriers are automatically insterted if e.g synchronized is used.)

Vectorization:

- Goal: improve performance by using specialized vector instructions
- SIMD: Single Instruction, applied to Multiple Data
- Requires vectorised code: code that uses the vector instructions provided by the target platform (CPU)
- Compilers(C++, JVMs JIT,...) attempt to detect vectorization opportunities → fully automated, but little or no control over if/where/how
- platform specific libraries (intrinsics,C/C++) expose vector instructions to developers → manual effort, but full control
- Poses no (additional) safety risk to concurrency

Instruction Stream : Instructions given to the CPU to execute

Instruction Level Parallelism (ILP)

- Goal: improve CPU performance by internal parallelisation
- CPU/Core detects independent operations in its instruction stream
- These may be executed in parallel inside the CPU if enough functional units (e.g floating-point unit,...) are available
- Various measures to increase potential for instruction parallelization. E.g speculatively execute instructions in parallel, even if result may not be used
- Concurrency hazard: cores only locally consider dependencies in their instruction stream, not globally across all cores. (Java e.g synchronized automatically adds memory barriers to prevent problematic reordering)
- Compilers may also reorder instructions; similar problems, same solution

0.3.1 Pipelining

Balanced Pipeline All steps require the same time

Throughput The amount of work that can be done by a system in a given period of time (How much can go through the pipeline in a given time)

- In CPUs : # of instructions completed per second
- The larger the throughput the better

Throughput bound $= \frac{1}{max(computationtime(stages))}$
1:= unit of work (e.g one instruction, one network package, ...)
max(computationtime(stages)) := the time of the longest step in the pipeline
The bound gives the throughput when the pipeline is at full utilization i.e it ignores lead-in and lead-out time

Latency Time needed to perform a given computation (I.e how long does it take one item to go through the pipeline)

- In CPU: time required to execute a single instruction in the pipeline
- Lower is better
- Pipeline latency is only constant over time if the pipeline is balanced (i.e each step takes the same time)
- more input means our bound is less exact

Latency bound $= \#stages \cdot max(computationtime(stages))$

Optimizing an unbalanced pipeline (E.g Clothes Washing) w: 5s d:10s f: 5: c:10, the given pipeline is unbalanced because drying and putting clothes in the closet takes more than the washing and folding. An attempt to balance the pipeline to get a constant latency would be to artificially increase the length of all steps to 10s, but in this case we would decrease the throughput. The other option is to add additional functional units i.e another dryer and closet increasing the total number of steps from 4 to 6 w:5s d1:4s d2:6 f:5 c1:4 c2:6, we then increase the duration of all steps to the duration of the longest step i.e 6, hence the pipeline is balanced and the throuput increased.

Throughput vs Latency Pipelining typically adds constant time overhead between individual stages (synchronization,communication), hence infinitely small pipeline steps are not practical and time it takes to get one complete task through the pipeline may take longer than with a serial implementation.

0.4 Basic Concepts in Parallelism

Expressing Parallelism The goal is to split up work of a single program into parallel tasks. This can be done Explicitly/Manually(task/thread parallelism) or Implicitly i.e Done automatically by the system (user expresses an operation and the system does the rest.

Work Partitioning & Scheduling

- work partitioning (task/thread decomposition)
  - split up work into parallel tasks/threads
  - done by user
  - A task is a unit of work
  - number of parititions should be larger than the number of processors
- scheduling
  - assign tasks to processors
  - typically done by the system
  - goal is full utilization i.e no processor is ever idle

Coarse vs Fine granularity

- Fine granularity
  - more portable (can be executed in machines with more processors
  - better for scheduling
  - but: if scheduling overhead is comparable to a single task → overhead dominates
- Task granularity guidelines
  - As small as possible but, significantly bigger than scheduling overhead

Scalability An overloaded concept: e.g how well a system reacts to increased load, for example clients in a server. In parallel programming:

- speedup when we increase processors
- what happens if $\#processors \rightarrow \quad \infty$
- program scales linearly → linear speedup

Parallel Performance Sequential execution time: T_1
Execution time T_p on p CPUs

- $T_p = T_1/p$ (Perfect Case)
- $T_p > T_1/p$ (Performance loss,what normally happens)
- $T_p < T_1/p$ (Can happen but unusual)

Parallel Speedup Speedup S_p on p CPUs $S_p = T_1/T_p$:

- $S_p = p$ linear speedup (Perfect Case)
- $S_p < p$ sub-linear speedup (Performance loss,what normally happens)
- $S_p > p$ super-linear speedup (Can happen but unusual)

Speedup is not only dependant on the program but also on the input.
Why $S_p < p$?
Programs may not contain enough parallelism (some parts might be sequential)
Overheads introduced by parallelization (typically associated with synchronization)
Architectural limitations (e.g memory contention)

Efficiency S_p/p how efficient is a multicore system for a given task

Amdahl's Law Execution time T_1 of a program falls into two categories:

- Time spent doing non-parallelizable serial work
- Time spent doing parallelizable work

Denoted: W_{ser}, W_{par}
Given P workers available to do parallelizable work, the times for sequential execution and parallel execution are:
$T_1 = W_{ser} + W_{par}$
Resulting in a bound on speed up: $T_p \geq W_{ser} + \frac{W_{par}}{P}$
$\Rightarrow$ Amdahls Law: $S_p \leq \frac{W_{ser}+W_{par}}{W_{ser}+\frac{W_{par}}{P}}$
We define f as the non-parallelizable serial fractions of the total work. The following equalities hold:

- $W_{ser} = fT_1$
- $W_{par} = (1-f)T_1$

$\Rightarrow S_p \leq \frac{1}{f+\frac{1-f}{P}} \quad \Rightarrow S_\infty \leq \frac{1}{f}$

Gustafson's Law Observations:

- consider problem size
- run-time, not problem size, is constant
- more processors allows to solve larger problems in the same time
- parallel part of a program scales with the problem size

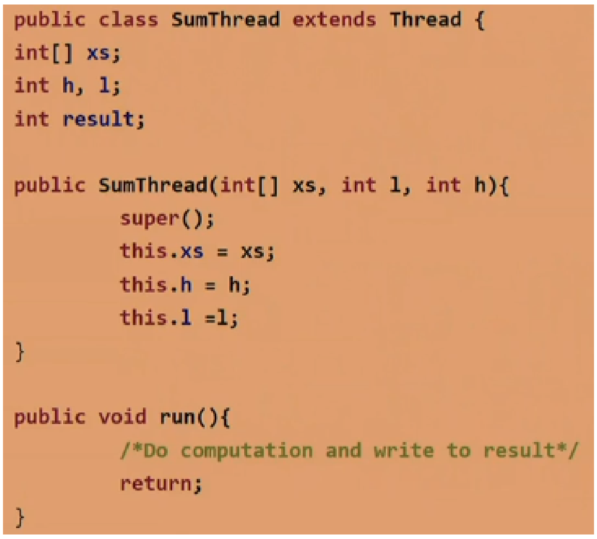
f : sequential part, $T_{wall} = availabletime$
$W = p(1-f)T_{wall} + fT_{wall}$
$S_p = \frac{S_p}{S_1} = f + p(1-f) = p - f(p-1)$

0.5 Divide and Conquer

fork/join Style of programming using start,run,join methods. They create a "happens before before relation", the ordering of the memory access is important and must be considered.

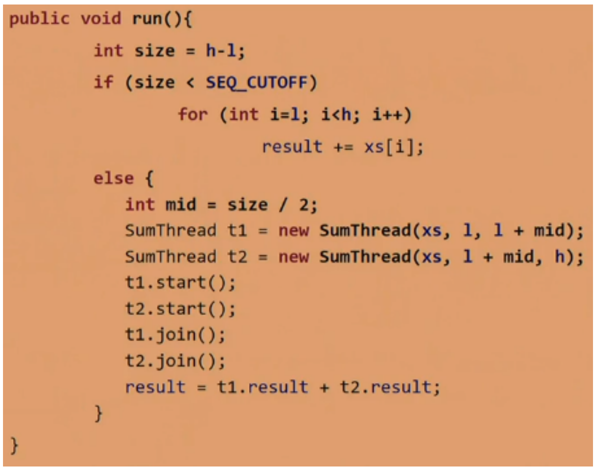
Approach to Divide and Conquer In theory you can divide down to single elements, do all your resultcombining in parallel and get optimal speedup. In practice, creating all those threads and communicating swamps the savings hence:

- Use a sequential cutoff, typically around 500-1000 (eliminates almost all the recursive thread creation(bottom levels of tree)
- Do not create two recursive threads, create one and do the other "yourself"
- If given enough processors, total time is height of the tree $\mathcal{O}(logn)$
- Often relies on operations being associative



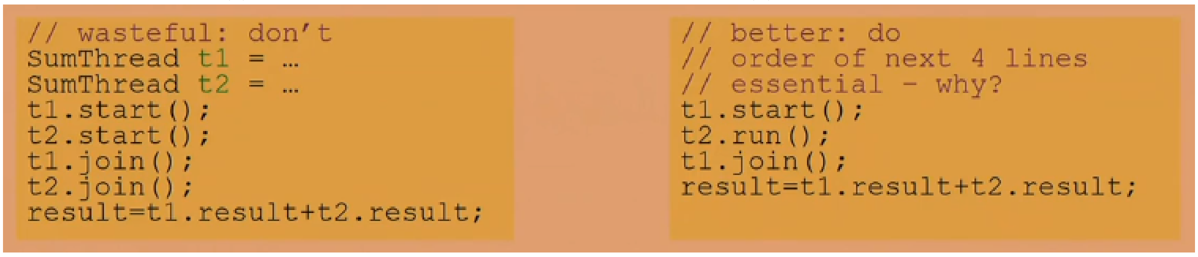
```
public class SumThread extends Thread {
int[] xs;
int h, l;
int result;

public SumThread(int[] xs, int l, int h){
        super();
        this.xs = xs;
        this.h = h;
        this.l =l;
}

public void run(){
        /*Do computation and write to result*/
        return;
}
```

(a) creating task



```
public void run(){
        int size = h-l;
        if (size < SEQ_CUTOFF)
                for (int i=l; i<h; i++)
                        result += xs[i];
        else {
           int mid = size / 2;
           SumThread t1 = new SumThread(xs, l, l + mid);
           SumThread t2 = new SumThread(xs, l + mid, h);
           t1.start();
           t2.start();
           t1.join();
           t2.join();
           result = t1.result + t2.result;
        }
}
```

(b) creating executer,submitting



```
// wasteful: don't
SumThread t1 = …
SumThread t2 = …
t1.start();
t2.start();
t1.join();
t2.join();
result=t1.result+t2.result;
```

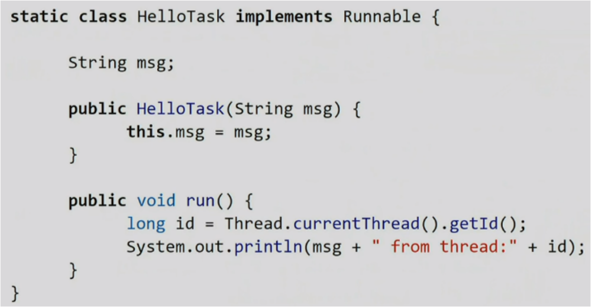
```
// better: do
// order of next 4 lines
// essential – why?
t1.start();
t2.run();
t1.join();
result=t1.result+t2.result;
```

(c) creating executer,submitting

Executor Service: Manages asynchronous tasks. ExecutorService is a Java Class which takes in a users submitted task and returns a "Future" object. Two ways to submit a task to the ExecutorService:

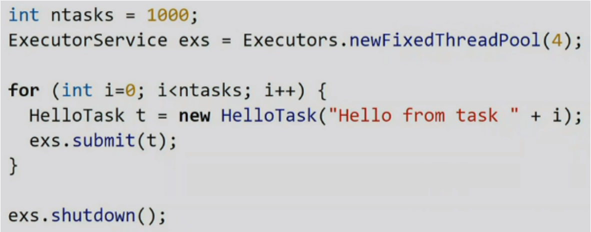
- .submit(Callable $< T >$ task) → Future$< T >$ (Returns result)
- .submit(Runnable task) → Future$<? >$ (Does not return result)

Beispiel:



```
static class HelloTask implements Runnable {

        String msg;

        public HelloTask(String msg) {
                this.msg = msg;
        }

        public void run() {
                long id = Thread.currentThread().getId();
                System.out.println(msg + " from thread:" + id);
        }
}
```

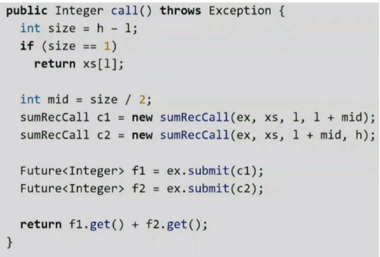
(a) creating task



```
int ntasks = 1000;
ExecutorService exs = Executors.newFixedThreadPool(4);

for (int i=0; i<ntasks; i++) {
  HelloTask t = new HelloTask("Hello from task " + i);
  exs.submit(t);
}

exs.shutdown();
```

(b) creating executer,submitting

Recursive Sum with ExecutorService:



```
public Integer call() throws Exception {
  int size = h - l;
  if (size == 1)
    return xs[l];

  int mid = size / 2;
  sumRecCall c1 = new sumRecCall(ex, xs, l, l + mid);
  sumRecCall c2 = new sumRecCall(ex, xs, l + mid, h);

  Future<Integer> f1 = ex.submit(c1);
  Future<Integer> f2 = ex.submit(c2);

  return f1.get() + f2.get();
}
```

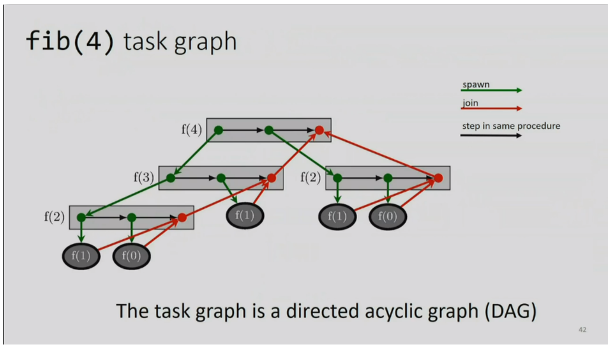
Figure 4

The get method blocks until the method which the Future object refers to is finished. The above implementation does not work because the ExecutorService is bound to a certain number of threads, hence we will eventually run out of threads and the tasks will end up waiting. The ExecutorService is not meant to be used when you need to wait for results of other tasks (divide and conquer). A possible approach is to decouple work partitioning from solving the problem. We split the array into chunks and create a task per chunk, we submit these into the ExecutorService and combine the results. When one task is finished the thread is freed and assigned to another task. I.e flat patterns (threads arent waiting) are good for the ExecutorService.

Cilk-style: Tasks:

- execute code
- spawn other tasks
- wait for results from other tasks

A graph is formes based on spawning tasks. There is an edge from node u to node v if task v was created by task u. Source vertice must finish first before destination starts. With Cilk there is no waiting for a certain task, but instead we wait for all tasks created until now to complete. There are no deadlocks in Cilk style programming (The task graphs are directed acyclic graphs).



fib(4) task graph

The task graph is a directed acyclic graph (DAG)

Figure 5

Task Parallelism:

- Tasks can execute in parallel, but they dont have to. The assignment of tasks to CPUs/Cores is up to the scheduler
- The task graph is dynamic and unfolds as execution proceeds(input dependent). A wide task graph means more parallelism

Performance Model: Tasks become available as computation progresses. We can execute the graph on p processors, the scheduler assigns tasks to the processors, hence the execution time T_p can vary depending on the scheduler being used.

- T_p execution time on p processors
- T_1 work(total amount of work) i,e the sum of the time cost of all nodes in graph (as if we executed graph sequentially)
- $\frac{T_1}{T_p}$ → speedup
- T_∞ span, critical path,computational depth: Time it takes on infinite processors i.e the longest path from root to sink
- $\frac{T_1}{T_\infty}$ → parallelism i.e maximum possible speedup
- Lower bounds:
  - $T_p \geq \frac{T_1}{p}$
  - $T_p \geq T_\infty$
- $T_p \approx \frac{T_1}{p} + T_\infty$

Scheduler is an algorithm for assigning tasks T_p depends on the the scheduler. $\frac{T_1}{P}$ and T_∞ are fixed The above figure shows that different schedulers can have different T_p. The boxes represent what is scheduled in each step.

0.6 ForkJoin Framework & Task Parallel Algorithms

ForkJoin Framework: Designed to meet the needs of divide-and-conquer fork-join parallism.

- .fork() → create a new task (Computation Graph: Ends a node and makes two outgoing edges i.e new thread and continuation of current thread)
- .join() → return result when task is done (Computation Graph: Ends a node and makes a node with two incoming edges i.e task just ended last node of thread joined on)
- .invoke() → submits task and waits until it is completed
- .submit() → submits task (recieves a Future)

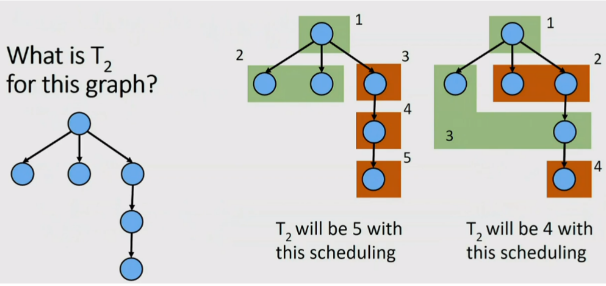
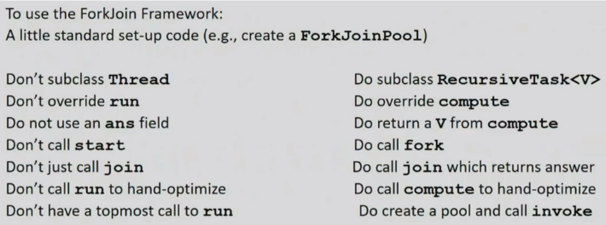

What is T_2 for this graph?
1
2
3
4
5
T_2 will be 5 with this scheduling
1
2
3
4
T_2 will be 4 with this scheduling

Figure 6

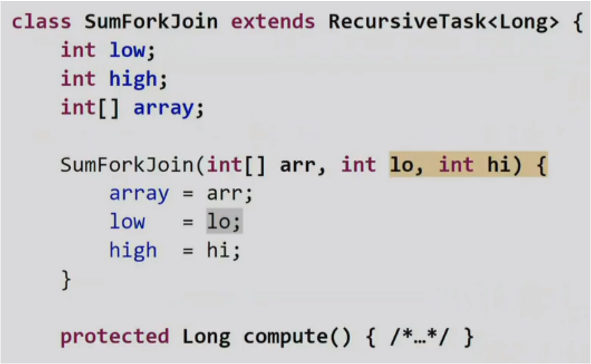


To use the ForkJoin Framework:
A little standard set-up code (e.g., create a **ForkJoinPool**)

Don't subclass **Thread**	Do subclass **RecursiveTask<V>**
Don't override **run**	Do override **compute**
Do not use an **ans** field	Do return a **V** from **compute**
Don't call **start**	Do call **fork**
Don't just call **join**	Do call **join** which returns answer
Don't call **run** to hand-optimize	Do call **compute** to hand-optimize
Don't have a topmost call to **run**	Do create a pool and call **invoke**

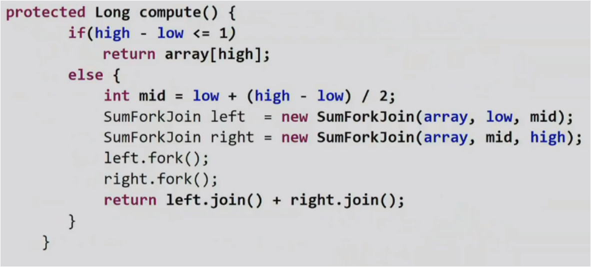
Figure 7

Recursive sum with ForkJoin: The ForkJoinPool creates a number of threads equal to the number of available processors.



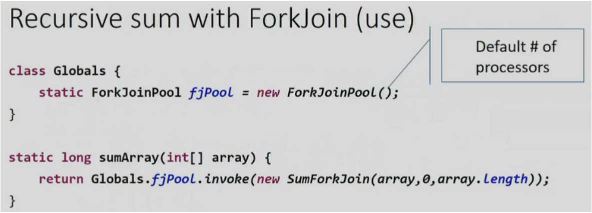
```
class SumForkJoin extends RecursiveTask<Long> {
    int low;
    int high;
    int[] array;

    SumForkJoin(int[] arr, int lo, int hi) {
        array = arr;
        low   = lo;
        high  = hi;
    }

    protected Long compute() { /*…*/ }
```

(a)



```
protected Long compute() {
        if(high - low <= 1)
            return array[high];
        else {
            int mid = low + (high - low) / 2;
            SumForkJoin left  = new SumForkJoin(array, low, mid);
            SumForkJoin right = new SumForkJoin(array, mid, high);
            left.fork();
            right.fork();
            return left.join() + right.join();
        }
    }
```

(b)



Recursive sum with ForkJoin (use)

Default # of processors

```
class Globals {
    static ForkJoinPool fjPool = new ForkJoinPool();
}

static long sumArray(int[] array) {
    return Globals.fjPool.invoke(new SumForkJoin(array,0,array.length));
}
```

(c)

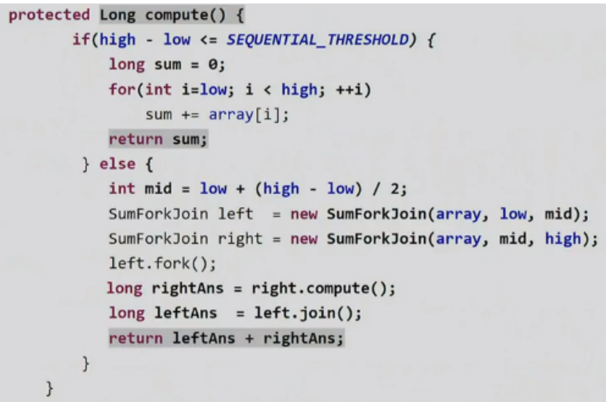
The code aboce performs poorly in java. The following fix is possible:

Reductions: Produce a single answer from collection via an associative operator (e.g max, count, leftmost,rightmost,...) (non examples: median, subtraction, exponentiation). (Recursive) results dont have to be a single number or strings. They can be arrays or objects with multiple fields. (e.g Histogram of test results is a variant of sum). But some things are inherently sequential i.e how we process arr[i] may depend entirely on the result of processing arr[i-1].

Maps: A map operates on each element of a collection independently to create a new collection of the same size, hence there is no combining results.

When to use Maps or Reduction:

- Data structure matters!
- Parallelism is still beneficial for expensive per-element operations on a sequential Datastructure (e.g Linked Lists)
- For parallelism, balanced trees are generally better than lists so that we can get to all the data exponentially faster $\mathcal{O}(logn)$ vs $\mathcal{O}(n)$

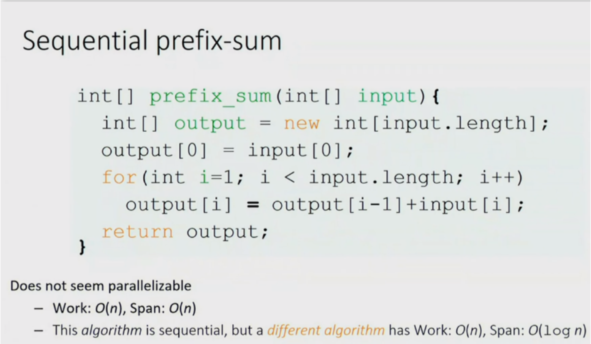


```
protected Long compute() {
        if(high - low <= SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for(int i=low; i < high; ++i)
                sum += array[i];
            return sum;
        } else {
            int mid = low + (high - low) / 2;
            SumForkJoin left  = new SumForkJoin(array, low, mid);
            SumForkJoin right = new SumForkJoin(array, mid, high);
            left.fork();
            long rightAns = right.compute();
            long leftAns  = left.join();
            return leftAns + rightAns;
        }
    }
```

Figure 9

The prefix-sum problem: Example used to show that inherently sequential programs can in fact be made parallel. Problem Statement:

- Given int[] input
- Produce int[] output
- output[i] = input[0] + input[1] + ... + input[i]



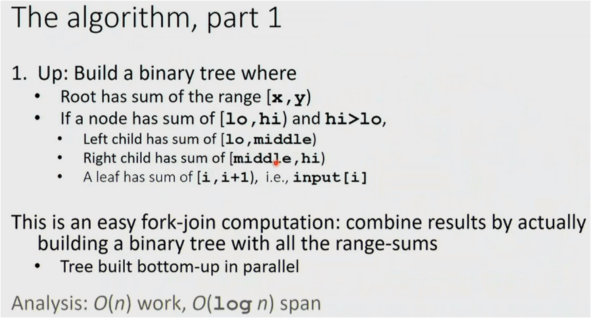
Sequential prefix-sum

```
int[] prefix_sum(int[] input){
  int[] output = new int[input.length];
  output[0] = input[0];
  for(int i=1; i < input.length; i++)
    output[i] = output[i-1]+input[i];
  return output;
}
```

Does not seem parallelizable

- Work: $O(n)$, Span: $O(n)$
- This *algorithm* is sequential, but a *different algorithm* has Work: $O(n)$, Span: $O(\log n)$

(a)



The algorithm, part 1

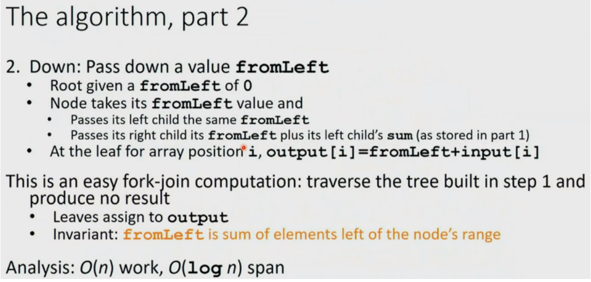
1. Up: Build a binary tree where
   - Root has sum of the range `[x,y)`
   - If a node has sum of `[lo,hi)` and `hi>lo`,
     - Left child has sum of `[lo,middle)`
     - Right child has sum of `[middle,hi)`
     - A leaf has sum of `[i,i+1)`, i.e., `input[i]`

This is an easy fork-join computation: combine results by actually building a binary tree with all the range-sums

- Tree built bottom-up in parallel

Analysis: $O(n)$ work, $O(\log n)$ span

(b)



The algorithm, part 2

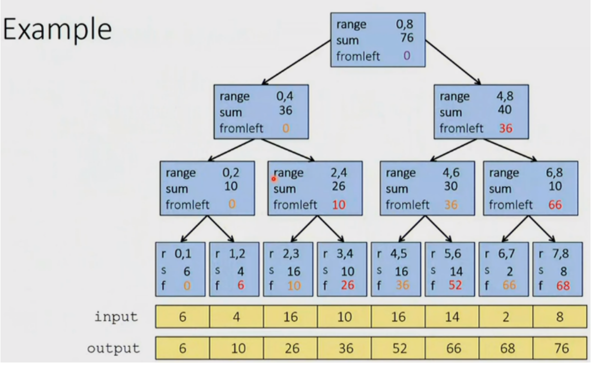
2. Down: Pass down a value **`fromLeft`**
   - Root given a **`fromLeft`** of 0
   - Node takes its **`fromLeft`** value and
     - Passes its left child the same **`fromLeft`**
     - Passes its right child its **`fromLeft`** plus its left child's **`sum`** (as stored in part 1)
   - At the leaf for array position **`i`**, **`output[i]=fromLeft+input[i]`**

This is an easy fork-join computation: traverse the tree built in step 1 and produce no result

- Leaves assign to **`output`**
- Invariant: **`fromLeft`** is sum of elements left of the node's range

Analysis: $O(n)$ work, $O(\log n)$ span

(a)



Example

input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76

(b)

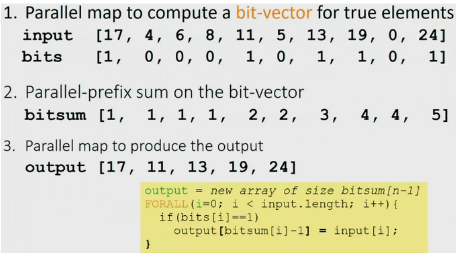
We get a parallel speedup at the expense of using more memory.

Pack Problem: Given an array input, produce an array output containing only elements such that f(elt) is true (i.e elements such that some property holds e.g elt ¿ 10). How is this Parallelizable? The work is $\mathcal{O}(n)$. Difficulty arises when trying to find the position of the current element in the result as its position depends on how many elements before it satisfy the condition. Solution (Using condition elt ¿ 10):

0.7 Shared memory concurrency, locks and data races

Managing State

- Immutability Data does not change. This is the best option and should be used when possible
- Isolated Mutability Data can change, but only one thread/task can access them



1. Parallel map to compute a bit-vector for true elements

```
input  [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits   [1,  0, 0, 0,  1, 0,  1,  1, 0,  1]
```

2. Parallel-prefix sum on the bit-vector

```
bitsum [1,  1, 1, 1,  2, 2,  3,  4, 4,  5]
```

3. Parallel map to produce the output

```
output [17, 11, 13, 19, 24]
```

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[i];
}
```

Figure 12

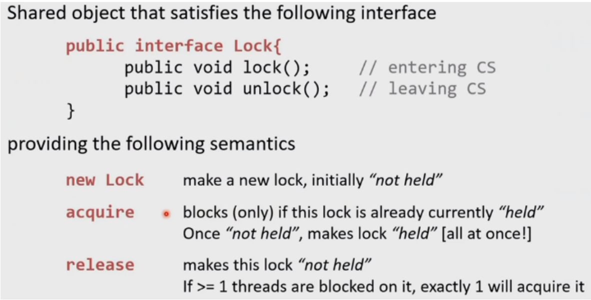
- Mutable/Shared data Data can change, multiple Tasks/Threads can potentially access the data

Mutable/Shared data: This is present in shared memory architectures. Concurrent accesses may lead to inconsistencies, hence we must protect the state by allowing only one thread/task access the memory at a time. We can achieve this by using the following methods:

- Locks: Mechanism to ensure exclusive access/atomicity (Assume that there will be other threads that will try to modify the memory)
- Transactional memory: Programmer describes a set of actions that need to be atomic (Perform actions and only after completion do we check if there was a conflict, if a conflict occured we rollback)

Mutual Exclusion: When one thread uses a resource another thread must wait until its free. (The resource is known as a critical section). Implementing critical sections is done by the programmer as the compiler is not capable of recognizing them and bad interleavings can occur.

Lock Object in Java: Locks ensure that given simultaneous acquires and/or releases, a correct thing will happen. Class for Reentrant Locks: java.util.concurrent.locks.ReentrantLock



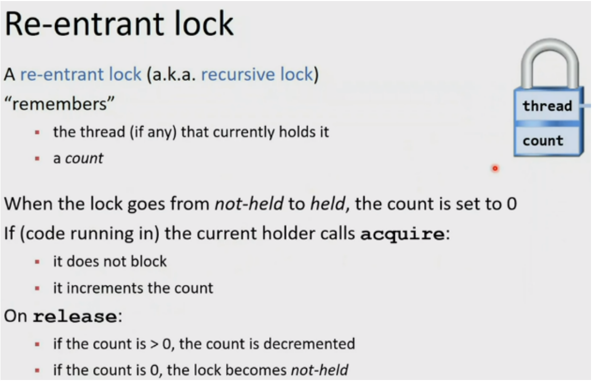
Shared object that satisfies the following interface

```
public interface Lock{
      public void lock();     // entering CS
      public void unlock();   // leaving CS
}
```

providing the following semantics

new Lock	make a new lock, initially *"not held"*
acquire	blocks (only) if this lock is already currently *"held"* Once *"not held"*, makes lock *"held"* [all at once!]
release	makes this lock *"not held"* If >= 1 threads are blocked on it, exactly 1 will acquire it

(a)



Re-entrant lock

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

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

When the lock goes from *not-held* to *held*, the count is set to 0

If (code running in) the current holder calls **acquire**:

- it does not block
- it increments the count

On **release**:

- if the count is > 0, the count is decremented
- if the count is 0, the lock becomes *not-held*

(b)

Races: A race condition occurs when the computation result depends on the scheduling (how threads are interleaved). There is no interleaved scheduling with only one thread but interleaved scheduling with only one processor is possible.

Data Race vs. Bad Interleaving:

- **Data Race:** [aka Low Level Race Condition] Erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads e.g Simultaneous read/write or write/write of the same memory location.
- **Bad Interleaving:**[aka High Level Race Condition] Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm that makes use of otherwise well synchronized resources.

3 options to avoid data races: For every memory location in your program, you must obey atleast one of the following:

- Thread-Local: Do not use the location for more than 1 threas
- Immutable: Do not write to the memory location
- Synchronized: Use synchronization to control access to the location

Thread-Local: Whenever possible, do not share resources.

- It is easier to have each thread have its own thread-local copy of a resource than to have one with shared updates
- This is only correct if threads do not need to communicate through the resource

- Because each call-stack is thread-local we do not need to synchronize on local variables

Immutable: Whenever possible do not update objects, instead make new objects. This helps to avoid side-effects and helps in a concurrent setting. If a location is read only then no synchronization is necessary (simultaneous reads are not races and not a problem.

The Rest: After minimizing the amount of memory that is thread-shared and mutable, we need guidelines for how to use locks to keep other data consistent. Guidlines:

1. No data races: Never allow two threads to read/write or write/write to the same location at the same time and do not make any assumptions on the orders of reads or writes.
2. Consistent Locking: For each location needing synchronization, have a lock that is always held when reading or writing the location. The lock "guards" the location and the same lock can guard multiple locations. (It is important to clearly document the guard for each location). Consistent locking is not sufficient, it prevents all data races but still allows bad interleavings.
3. Lock granularity: Start with coarse-grained and move to fine-grained only if contention on the coarser locks becomes an issue.
    - Coarse-grained: Fewer locks i.e more objects per lock (e.g one lock for an array). Coarse grained locking is simpler to implement and faster/easier to implement operations that access multiple locations. Also much easer to implement operations that modify the data-structures shape.
    - Fine-grained: More locks i.e fewer objects per lock (e.g one lock per array index). Fine grained locking allows for a more simultaneous access (performance when coarse-grained would lead to unnecessary blocking)
4. Critical-section granularity: Do not do expensive computations or I/O in critical sections, but also dont introduce race conditions. A second orthogonal granularity issue is critical section size. If the critical sections run for too long then performance will be lost because of other threads being blocked. On the other hand if critical sections are too short then bugs can be created because other threads see intermediate states they shouldnt and performance can be lost because of frequent thread switching and cache trashing.
5. Atomicity: Think in terms of what operations need to be atomic. An operation is atomic if no other thread can see it partly executed ("appears" invisible). Make the critical sections just long enough to preserve atomicity, then design the locking protocol to implement the critical sections correctly i.e Think about atomicity first and locks second.

Memory Reordering: The Compiler and hardware are allowed to make changes that do not affect the semantics of a sequentially executed program. What gets reordered depends on hardware e.g AMD86 is different than ARM.

- Software view: Modern compilers do not give guarantees that a global ordering of memory accesses is provided. Some memory accesses may be optimized away completely.
- Hardware view: Modern multiprocessors do not enforce global ordering of all instructions because of performance gains. Most processors have a pipelined architecture and can execute multiple instructions simultaneously. They can (and will) reorder instructions internally. Each processor has a local cache, and thus loads/stores to shared memory can become visible to other processors at different times.

There are some language constructs that forbid such reordering. (in Java synchronized and volatile)

Memory Models: The exact behaviour of threads interacting via shared memory usually depends on hardware,runtime system, and programming language. A memory model provides guarantees for the effects of memory operations, leaving open optimization possibilities for hardware and compiler, but including guidelines for writing correct multithreaded programs.

Java Memory Model(JMM) :

- JMM restricts allowable outcomes of programs
- JMM defines Actions: read/write e.g read(x):1 "read variable x, the value read is 1"
- Exections combine actions with ordering:
    - Program Order (Order in which statements are executed)
    - Synchronizes-with (Order of observed synchronizing memory actions across threads)
    - Synchronization Order (oder of synchronizing memory actions in the same thread)
    - Happens-before (union(transitive closure) of PO and SW)

Program Order(PO):

- Program order is a total order of intra-thread actions. Program statements are NOT a total order across threads.
- Program order does not provide an ordering guarantee for memory accesses
- Intra-thread consistency: Per thread, the PO order is consistent with the threads isolated execution

Synchronization Actions(SA):

- Read/write of a volatile variable
- Lock monitor, unlock monitor
- First/last action of a thread
- Actions which start a thread
- Actions which determine if a thread has terminated

underlineSynchronization Order(SO): formed by the synchronization actions:

- SO is a total order (all threads see the same order)
- all threads see SA in the same order
- SA within a thread are PO
- SO is consistent, all reads in SO see the last writes in SO

Synchronizes-With (SW)/ Happens-Before (HB) order:

- SW only pairs the specific actions which see eachother
- A volatile write to x synchronizes with subsequent read of x
- The transitive closure of PO and SW forms HB
- HB consistency: When reading a variable, we see either the last write in HB or any other unordered write

0.8 Behind Locks: Implementation of Mutual Exclusion

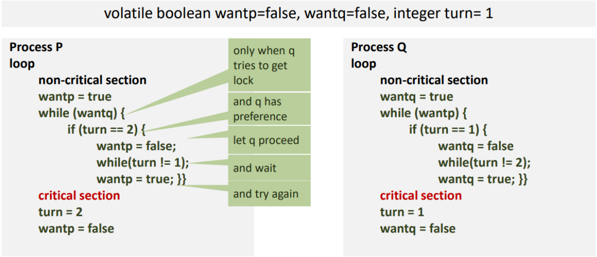
Assumptions:

- Atomic reads and writes of variables of primitive type
- no reordering of read and write sequences (not true in practice!)
- threads entering a critical section will leave it eventually
- otherwise we assume a multithreaded environment where processes can arbitrarily interleave
- we make no assumptions for progress in non critical secion

Critical Sections: Pieces of code with the following conditions:

- Mutual exclusion: statements from critical sections of two or more porcesses must not be interleaved
- Freedom from deadlock: if some processes are trying to enter a critical secion then one of them must eventually succeed
- Freedom from starvation: if any process tries to enter its critical section, then that process must eventually succeed

Decker's Algorithm:



```
volatile boolean wantp=false, wantq=false, integer turn= 1

Process P
loop
    non-critical section
    wantp = true
    while (wantq) {
        if (turn == 2) {
            wantp = false;
            while(turn != 1);
            wantp = true; }}
    critical section
    turn = 2
    wantp = false

Process Q
loop
    non-critical section
    wantq = true
    while (wantp) {
        if (turn == 1) {
            wantq = false
            while(turn != 2);
            wantq = true; }}
    critical section
    turn = 1
    wantq = false
```

only when q tries to get lock

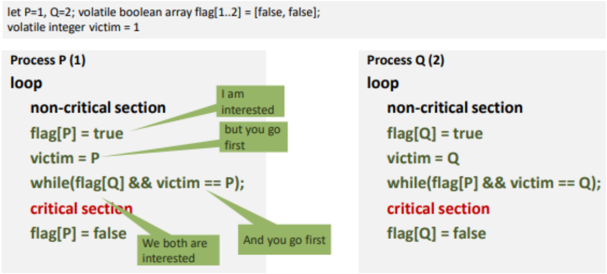
and q has preference

let q proceed

and wait

and try again

Peterson Lock: When implementing Peterson in Java setting an array to volatile doesnt work. Volatile will be the



```
let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

Process P (1)
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false

Process Q (2)
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

I am interested

but you go first

We both are interested

And you go first

reference to the array and not an array of volatile variables, instead we use Java's AtomicInteger and AtomicIntegerArray.

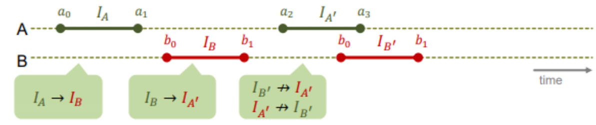
Events and precedence: Threads produce a sequence of events

$$\text{P produces events } p_0, p_1$$
$$\text{e.g } p_1 = \text{"flag[P] = true"}$$

j-th occurence of event i in thread P: p_i^j (e.g p_5^3 = "flag[P] = false" in the third iteration.)
Precedence relation: we write $a \to b$ when a occurs before b (the precedence relation is a total order for events)

Intervals:

$$(a_0, a_1) : \text{interval of events } a_0, a_1 \text{ with } a_0 \to a_1$$
$$\text{With } I_A = (a_0, a_1) \text{ and } I_B = (b_0, b_1) \text{ we write } I_A \to I_B \text{ if } a_1 \to b_0$$



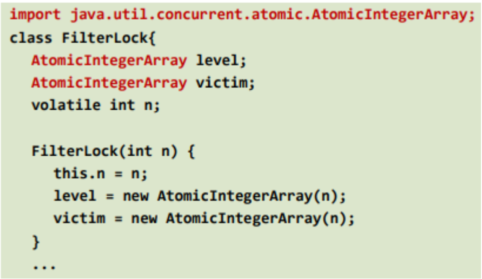
we say "I_A ***precedes*** I_B" and "$I_{B'}$ and $I_{A'}$ are *concurrent*"

Atomic register: A Register is a basic memory object which can be shared or not (i.e in this context register $\neq$ register of a CPU). A Register r has two operations: r.read() and r.write(v). Atomic Register has the following structure:

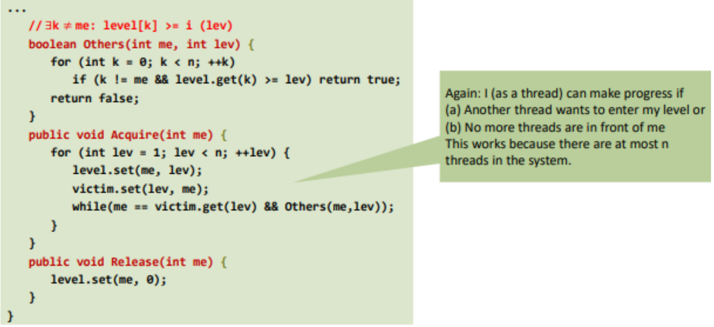
- An invocation J of r.read or r.write takes effect at a single point $\tau(J)$ in time (i.e no two reads or writes will happen simultaneously)
- $\tau(J)$ always lies between start and end of the operation J
- Two operations J and K on the same register always have a different effect time $\tau(J) \neq \tau(K)$

These assumptions for Atomic Registers justify to treat operations on them as events taking place at a single point in tiime. Even with atomic registers there can still be non determinism of programs because nothing is said about the order of effect times for concurrent operations.

Filter Lock: Extension of Peterson's lock to n processes. Every thread t knows his level in the filter level[t]. In order to enter CS a thread has to elevate all levels. For each level, we use Peterson's mechanism to filter at most one thread, if other threads are at higher level. For every level l there is one victim victim[l] that has to let others pass in case of conflicts.



```
import java.util.concurrent.atomic.AtomicIntegerArray;
class FilterLock{
   AtomicIntegerArray level;
   AtomicIntegerArray victim;
   volatile int n;

   FilterLock(int n) {
      this.n = n;
      level = new AtomicIntegerArray(n);
      victim = new AtomicIntegerArray(n);
   }
   ...
```



```
...
   //∃k ≠ me: level[k] >= i (lev)
   boolean Others(int me, int lev) {
      for (int k = 0; k < n; ++k)
         if (k != me && level.get(k) >= lev) return true;
      return false;
   }
   public void Acquire(int me) {
      for (int lev = 1; lev < n; ++lev) {
         level.set(me, lev);
         victim.set(lev, me);
         while(me == victim.get(lev) && Others(me,lev));
      }
   }
   public void Release(int me) {
      level.set(me, 0);
   }
}
```

Again: I (as a thread) can make progress if
(a) Another thread wants to enter my level or
(b) No more threads are in front of me
This works because there are at most n threads in the system.

Fairness: Divide lock implementation into two parts:

- Doorway interval D: finite number of steps
- Waiting interval W: unbounded number of steps

A lock algorithm is first-come-first-served when for two processes A and B holds that if $D_A^j \to D_B^k$ then $CS_A^J \to CS_B^k$