Midterm II

This document is not yet complete. Missing are the following:

- **Notes on exercises 7 (8)**

The missing parts might be added …. The document's version will then change from 1.2b to 1.3b

Table of Contents

Info

Study Part

The contents are structured according to the lecture slides. Some of the texts are 1:1 copies from the lecture slides available at http://ait.inf.ethz.ch/teaching/courses/2014-SS-Parallel-Programming/. For the sake of readability and quicker typing, those excerpts are often simply in double quotes. And sometimes they are simply paraphrased. All credit goes to Prof. Dr. O. Hilliges and Dr. K. Kourtis.

11 Synchronization: Beyond Locks

- Locks provide means to enforce atomicity via mutual exclusion yet they lack means for threads to communicate about changes
- Example: producer/consumer (p/c) queues (think: bakery): can be used for data-flow parallel programs, e.g. pipelines where a mean to transfer X from the producer to the consumer is needed. There might be multiple producers or (not xor) multiple consumers. For an implementation a circular buffer (with a fixed size) can be used with simple dequeue()/enqueue() and an "in" and "out" counter. Both functions use a shared (reentrant) lock and rely on helper functions to check for full/empty queue[1]. If you use a busy wait (while loop) there is a chance of a deadlock (and CPU running high). Using sleep for synchronization as another approach is generally discouraged[2]. The solution is a condition variable which (ideally) notifies the threads upon change.
- A condition interface provides the following methods: .await() – the current thread waits until it is signaled; .signal() – wakes up one waiting thread; .signalAll() – wakes up all waiting threads. Conditions are **always** associated with a lock. .await() is called with the lock held, releases the lock *atomically* and waits for thread to be signaled and is *guaranteed* to hold the lock when returning and the threads *always* needs to check the condition. .signal[All]() is called with the lock held.
- Check then act![3]
- Conditions can also be used with intrinsic locks where each object can act as a condition, implementing .notify(), .notifyAll(), .wait(). They do not allow for multiple conditions.
- Object.wait and Codition.await: always have a condition predicate; always test the condition predicate: before calling wait and after returning from wait; always call wait in a loop; ensure state is protected by lock associated with condition
- Semaphores[4] have the following operations: *initialize* to an integer value and after initialization only wait/signal operations are allowed; *acquire*: integer value is decreased by one, if < 0 → thread suspends execution; *release*: integer value is increased by one if there is at least a thread waiting, one of the waiting threads resume execution; A thread cannot know the value of a semaphore and there is no rule about what thread will continue its operation after a release()
- Say you build a lock (mutex) with a semaphore, you initialize it to 1 and then 1 means unlocked, 0 is locked, -n means n threads are waiting to enter.
- You can (of course) also use semaphore for p/c queues, however you need to use two semaphores to order the operations (and to prevent a deadlock).
- Barrier: rendezvous for arbitrary number of threads i.e. every thread has to wait up for all other threads to arrive at a certain point; can be implemented for n threads with two semaphores (and one count variable), one as a mutex (used to atomically increment the counter) with default = 1 and one as a barrier with default = 0 (which is released if count == n and otherwise only acts as acquire-and-release-immediately).

---

[1] Note: If you have a try-catch-finally block and there is a return statement (assume it will be called no matter what) in the "try" part and an unlock() in the "finally" part, the finally part **will always** be executed (and thus also the lock released!).
[2] Mentioned in an earlier lecture.
[3] This can also be helpful for bungee-jumping.
[4] Language background: semaphore is fancy for traffic light in English (see also Spanish).

- If you want a reusable barrier for n threads (aka 2-phase barrier) with semaphores, you need a count, a mutex and two barriers for it to be thread-safe.

12 Advanced (and other) Topics

- Locks can be implemented with low-level atomic operations and busy wait loops.
- Simple example: Peterson lock [for two threads] (see code on the right) where two AtomicBooleans (one per thread) and an AtomicInteger which decides which thread will be selected.
- Rich(er) atomic operations for AtomicInteger: getAndSet(val) (atomically { set to val, return old value }), getAndAdd(val), getAndIncrement, getAndDecrement, CompareAndSet (CAS for short)

```
AtomicBoolean t0 = new AtomicBoolean(false);
AtomicBoolean t1 = new AtomicBoolean(false);
AtomicInteger victim = new AtomicInteger(0);
lock:
my_t.set(true)
victim.set(me);
while (other_t.get() == true &&
victim.get() == me)
;
unlock:
my_t.set(false);
```

- Lock using getAndSet: mutex is an AtomicBoolean which is set to either true or false on lock or unlock (resp.)
- CAS(int old, int new): performs atomically the following (optimistically): if current_val == cold then current_val = new, return true else return false
- Lock using getAndSet: mutex is an AtomicBoolean which is either CAS'ed as compareAndSet(false, true) or set(false) on lock or unlock (resp.)
- Busy-waits check continuously for a value and waste CPU-time (or as alternative: exponential backoff) which should be avoided using a notification mechanism of sorts
- Mutexes: locks that suspend the execution of threads while they wait are typically called mutexes (vs spinlocks); scheduler (typically from the OS) support is required; they do not waste CPU time but they have higher wakeup latency; hybrid approach: spin and then sleep
- Locks performance: Uncontended case: when threads do not compete for the lock, lock implementations try to have minimal overhead, typically just the cost of an atomic operation; Contended case: when threads do compete for the lock, can lead to significant performance degradation, also, starvation
- Disadvantages of locking: locks are pessimistic by design, they assume the worse/worst and enforce mutual exclusion; performance issues: overhead for each lock taken even in uncontended case, contended case leads to significant performance degradation; blocking semantics (wait until acquire lock), if a thread is delayed for a reason (e.g., scheduler) when in a critical section → all threads suffer, lead to deadlocks (and also livelocks)
- *Locks*: a thread can indefinitely delay another thread; *non-blocking*: failure or suspension of one thread cannot cause failure or suspension of another thread; *Lock-free*: at each step, some thread can make progress
- Non-blocking algorithms: typically built using CAS (more powerful than plain-atomic); see lecture slides for stack example
- Overview of what java.util.concurrent has to offer
  - Lock interface with lock(), lockInterruptibly(), tryLock([delay][5]), unlock(), newCondition(); implemented by ReentrantLock
  - ReadWriteLock interface with readLock() writeLock(); implemented by ReentrantReadWriteLock; multiple readers can concurrently access state whereas writers get exclusive access, beneficial for scenarios with comparably few writes; can be implemented with semaphores but fairness might be an issue leading to starvation[6] unless prevented by means to notify the read lock about waiting writers

[5] Using the PHP docs style where square brackets denote optional arguments
[6] Starvation: when a particular thread cannot resume execution; different from deadlock, where all the threads are unable to

- Collections: objects that group multiple elements into a single unit; interfaces: Collection, List, Set, SortedSet, …; implementations: ArrayList, LinkedList, …; Algorithms: sort, …; based on Java generics
- Synchronized Collections: Vector, HashTable, synchronizedList, synchronizedMap, synchronizedSet, synchronizedSortedMap, synchronizedSortedMap, synchronizedCollection; they are wrapper classes, basically wrapping every public method in a synchronized block, they are thread safe but poor concurrency due to a single, collection-wide lock
- Concurrent Collections: thread safe, but not a single lock; ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArrayList
- Queues: BlockingQueue: ArrayBlockingQueue, LinkedBlockingQueue (FIFO), PriorityBlockingQueue (ordered); TransferQueue: allows to wait until a consumer receives item; SynchronousQueue: hand-of, no internal capacity; Dequeue/BlockingDeque: allows efficient removal/insertion at both ends (head/tail), work stealing pattern
- Synchronizers: Semaphores, CyclicBarrier, CountDownLatch (thread wait until countdown reaches zero)
- Future: interface with get(), isDone(), cancel() representing a result for async computation

13 Parallel Tasks

Example for most of this lecture: $\sum x_i$

```
public static int sum(int[] xs) {
 int sum = 0;
 for (int x: xs)
  sum += x;
 return sum;
}
```

- When writing a parallel program, write a sequential version first! This is useful for knowing the results are correct and evaluate the performance of the parallel program.
- Divide-and-conquer approach: recursive sum with the lower and upper half of the remaining part which cuts off at size = 1 is a lot slower (x10).[7]
- Task parallel model: basic operations: create a parallel task and wait for the tasks to complete; when using D&C a task for the first and second part (one each) are created and upon finishing their results are combined
- One thread per task: expensive to create, consumes many resources and is scheduled by the OS – generally inefficient
- ExecutorService: A (huge) amount of tasks is handled by an interface which assigns a thread from a thread pool to each task and returns a Future[8]
- Note: Runnable doesn't return a result, Callable does
- ExecutorService and recursive sum[9]: task is described as the array to be summed and the region for which the task is responsible for, additionally an instance to the ExecutorService is passed so that the task can spawn other tasks; problems (observation: no result returned): Tasks create other tasks and then wait for results, when they are waiting they are keeping threads busy, other tasks need to run so that the recursion reaches its bottom, system does not know that tasks waiting need to be removed so that other tasks can run *due to*: tasks create other tasks (which is not supported) and work partitioning (splitting up work) is part of the task – we can decouple work partitioning from solving the problem
- Fork/Join framework with ForkJoinTask (fork() creates a new task, join() returns the result when task is done, invoke() executes task without creating a new task; subclasses need to define compute()) implements Future and ForkJoinPool implements ExecutorService; Note fork(), fork(), join(), join() doesn't work (well) in Java, solved by using: t1.fork(), r2 = t2.compute(), return r2 + t1.join();[10]

---

proceed

[7] Code not listed since the code in the slides is (by design) naïve and simple, everyone should know how to do that by heart.

[8] See notes about previous lecture

[9] Have a look at the code in the slides, pp 36 – 38

[10] "+" is in this case the arithmetic addition but can also be something else of a combining nature

- Problems of overhead: bad speedup due to too much overhead (scheduling etc.), can be solved by making each task work more, here: increase cutoff

14 Transactional Memory (TM)

- Aims at removing the burden of having to deal with locks from the programmer and place it on the system instead
- Problems with locks: ensuring ordering and correctness is really hard and locks are not composable; locks are pessimistic, performance overhead; locking mechanism is hard-wired to the program (separation not possible and change of synchronization scheme results in changing all of the program)
- With TM, the programmer explicitly defines atomic code sections and is only concerned with the *what* and not the *how* (declarative approach)
- TM benefits: easier, less error-prone, higher semantics, composable, optimistic by design; changes made by a transaction are made visible atomically; transactions run in isolation – while a transaction is running, effects from other transactions are not observed (as if transaction takes a snapshot of the global state when it begins and operates on that snapshot);note: while locks enforce atomicity via mutual exclusion, transaction does not require that
- TM is inspired by transactions in databases where transactions are vital; ACID – Atomicity, Consistency, isolation, Durability
- Implementation of TM: Keep track of operations performed by each transaction: concurrency control, system ensures atomicity and isolation properties
- Transactions can be aborted if a conflict has been detected by the concurrency control (CC) mechanism; aborts are possible e.g. if there's a deadlock; on abort, a transaction can be retried automatically or the user is notified
- Where TM is/can be implemented: Hardware TM[11]: can be fast but cannot handle big transactions; Software TM (STM)[12]: in the language, greater flexibility, performance might be challenging; Hybrid TM; TM is still work in progress with many different approaches and us still under active development
- Design choice: strong vs weak isolation: **Q**: What happens when shared state accessed by a transaction, is also accessed outside of a transaction? Are the transactional guarantees still maintained? **A**: Strong isolation: Yes, easier for porting existing code, difficult to implement, overhead; Weak isolation: No
- Design choice: Nesting[13]: **Q**: What are the semantics of nested transactions? (Note: nested transactions are important for composability) **A**: flat nesting (inner aborts → outer aborts; inner commits → changes visibly only if outer commits), closed nesting (inner abort does not result in an abort for the outer transaction; inner transaction commits → changes visible to outer transaction but not to other transaction; only when outer transaction commits, changes of inner transactions become visible), other approaches (e.g. open nesting)
- The more variables are part of a transaction (and thus protected) the easier it gets to port existing code but the more difficult to implement ,too (need to check every memory operation)
- Reference-based STMs: mutable state is put into special variables; these variables can only be modified inside a transaction, everything else is immutable (or not shared; see functional programming)
- Mechanism of retry: implementations need to track what reads/writes a transaction performed to detect conflicts, typically called read-/write-set of a transaction; when retry is called, transaction aborts and will be
- retried when any of the variables that were read, change
- Issues with transactions: it is not clear what the best semantics for transactions are; getting good performance can be challenging; I/O operations: can we perform I/O operations in a transaction?

---

[11] Intel Haswell (4th generation i3/5/7 processors) is the first wide-spread implementation of hardware TM
[12] Haskell, Clojure, …
[13] See also https://www.facebook.com/groups/infstudents13/permalink/659004517480019

- I/O[14] in transactions: in general, I/O operations cannot be rolled-back and thus generally cannot be aborted; that is why I/O operations are not allowed in transactions; one of the big issues with using TM; (some) STMs allow registering I/O operations to be performed when the transaction is committed

15 Designing Parallel Algorithms

- There are no rules whatsoever, yet – as (very) often – it is a matter of experience
- The following points can/should be considered
  - Where do the basic units of computation (tasks) come from? This is sometimes called "partitioning" or "decomposition". Depending on the problem partitioning in terms of input and/or output can make sense or functional decomposition might yield better results
  - How do the tasks interact? We have to consider the dependencies between tasks (dependency, interaction graphs). Dependencies will be expressed in implementations as communication, synchronization and sharing (depending upon the machine model).
  - Are the natural tasks of a suitable granularity? Depending upon the machine, too many small tasks may incur high overheads in their interaction. Should they be collected together into super-tasks?
  - How should we assign tasks to processors? In the presence of more tasks than processors, this is related to scaling down. The "owner computes" rule is natural for some algorithms which have been devised with a data-oriented partitioning. We need to ensure that tasks which interact can do so as (quickly) as possible.
- D&C is a very important technique and particularly helpful in PP since the recursive step can instead be parallelized
- Number of threads to be used: "Runtime.getRuntime().availableProcessors();" *might* be the right amount but your program may not get access to all cores; too few threads are bad because core(s) is/are idle; too many threads can be bad because of the overhead[15]
- Sorting[16]: If the array is sorted the following condition must hold (equal only if $A_i = A_j$): $A_i \leq A_j \; for \; i < j$; features of a sorting algorithm: stable (duplicate data is allowed and the algorithm does not change duplicate's original ordering relative to each other), in-place ($O(1)$ auxiliary space), non-comparison; some sorting algorithms: horrible $\Omega(n^2)$: bogo, stooge; simple $O(n^2)$: insertion[17], selection[18], bubble, shell; fancier $O(n \log n)$: heap, merge, quick sort (**on average!**); specialized $O(n)$: bubble, radix
- Linked Lists and Big Data: Mergesort can very nicely work directly on linked lists; Heapsort and Quicksort do not; InsertionSort and SelectionSort can too but slower; Mergesort also the sort of choice for external sorting
- Quicksort and Heapsort jump all over the array; Mergesort scans linearly through arrays; In-memory sorting of blocks can be combined with larger sorts; Mergesort can leverage multiple disks
- **PRAM model**: processors working in parallel, each is trying to access memory values; when designing algorithms, the type of memory access required needs to be considered; scheme for naming different types: [concurrent|exclusive]READ[concurrent|exclusive]WRITE[19]; typically CR are not a problem since the memory isn't changed whereas EW requires code to ensure writing is exclusive; PRAM is helpful to envision how it works and the needed data access pattern but isn't necessarily the way processors are arranged in practice

[14] send/receive data over the network, write data to disks, push a button to launch a missile; essentially escape the CPU & memory system
[15] This depends on the actual overhead the language introduces (in Java rather big)
[16] This is D&A thus not covered to its full extent
[17] At step k, put the k^{th} input element in the correct position among the first k elements
[18] At step k, find the smallest element among the unsorted elements and put it at position k
[19] Abbreviated as E/C and R/W; ERCW is never considered

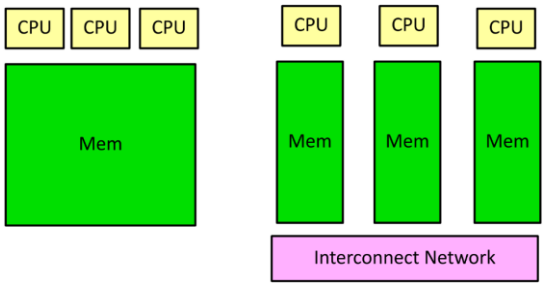
16 Java GUIs – MVC – Parallelism

Don't get me wrong, but I'm having a hard time writing up this lecture…

- (important) concepts: MVC (model (application domain, state and behavior) – view (display layout and interaction views) – controller (user input, device interaction)); layout managers; event-driven design[20] (listener, worker[21], callback, fire/handle), GUI (painting)
- Swing threads: initial[22], event dispatch[23] and worker thread[24]
- **MVC**: *model*: complete, self-contained representation of object managed by the application, provides a number of services to manipulate the data, computation and persistence issues; *view*: tracks what is needed for a particular perspective of the data, presentation issues; *controller*: gets input from the user, and uses appropriate information from the view to modify the model, interaction issues

17 Concurrent Message Passing

- goal: avoid (mutable) data sharing, instead use concurrent message passing (actor programming model) since many of the PP problems (so far) are due to shared state
- isolated mutable state: state is mutable, but not shared; each thread/task has its private state; tasks cooperate with message passing
- shared memory architecture (left side in image): message passing and sharing state is used; message passing: can be slower than sharing data yet is easier to implement and to reason about
- distributed memory architecture (right side in image):: sharing state is challenging and often inefficient, using almost exclusively message passing; additional concerns such as failures



- message passing works in both shared and distributed memory architectures making it more universal
- example: shared state counting (i.e. atomic counter) with increase() and get(): approach #1: one counter thread, the other threads ask for its value; approach #2: every thread has its own (local) counter (Java: ThreadLocal), when sum is requested all threads return the value of their local counter
- example: bank account: sequential programing: single balance; PP shared state: single balance & protection; PP distributed state: each thread has a local balance (budget), threads share balance coarsely
- distributed bank account (cont.): each task can operate independently, only communicate when needed
- synchronous vs asynchronous messages: sync: send blocks until message is received (Java: SynchronousQueue); async: send does not block ("fire-and-forget"), placed into a buffer for receiver to get (Java: BlockingQueue, async as long as there is enough space (to prevent memory overflow))
- concurrent message passing programming models: actors: state-full tasks communicating via messages (e.g. erlang); channels[25]: can be seen as a level of indirection over actors, Communicating Sequential Process (CSP) (e.g. go)
- go (by Google): language support for: lightweight tasks (aka goroutines), typed channels for task communications which are synchronous (unbuffered) by default
- actor programming model: a program is a set of actors that exchange (async) messages; actor embodies: state, communication, processing
- An actor may: process messages, send messages, change local state, create new actors

---

[20] E.g. agents in Eiffel; .on() in jQuery, …
[21] In Swing, this implements Runnable
[22] Main thread
[23] Drawing/painting the GUI
[24] Background thread, can be used for (heavy) computation to keep GUI responsive
[25] not an official term

- event-driven programming model: a program is written as a set of handlers (typical application: GUI)
- erlang: functional language; developed for fault-tolerant applications, if no state is shared, recovering form errors becomes much easier; concurrent, following the actor model; open-source
- actor example: distributor: forward received messages to a set of names in a round-robin fashion: state: an array of actors with the array index of the next actor to forward a message; receive: messages -> forward message and increase index (mod), control commands (e.g. add/remove actors)
- actor example: serializer: unordered input (e.g. due to different computation speed) -> ordered output; state: sorted list of received items, last item sent; receive: if we receive an item that is larger than the last item plus one, add it to the sorted list; if we receive an item that is equal to the past item plus one: send the received item plus all consecutive items form the last and reset the last item
- concurrent message passing in Java_ for simple applications, queues can be used which might be difficult especially for large tasks; instead use akka framework (written in Scala, interface for Java): follows the actor model (async messages), rich set of features[26]
- akka actors example: ping-pong: client sends n PINGs to server which responds with Pong upon receiving back to sender, master stops execution when receiving DONE[27]; version 2 with restart on DONE: add a message type SETUP to the client passing the server actor reference and the count, if the client receives SETUP before DONE it can either wait for DONE and the restart or discard the message
- collective operations: *broadcast*: send a message to all actors (related: multicast, sending a message to some actors), parallel broadcast using a tree where every parent forwards the message to its children until it reaches the leafs (top-down); *reduction*: perform a computation from values of multiple nodes (e.g. balance of all bank accounts), using a tree where a parent receives the message from its children, performs operation and sends it to parent (bottom-up)

18 Data Parallel Programming

Data Parallel Programming

- task vs data parallelism: task: work is split into parts, by parallelizing the algorithm, very generic but cumbersome; data: simultaneously applied operation on an aggregate of individual items (e.g. array), declarative (= what not how), splitting up the data for parallelism, less generic
- main operations in data parallelism: map, reduce, prefix scan, parallel loop
  - **map**: input: array (x), operation ($f(\cdot)$); output: aggregate with applied operation ($f(x)$); parallel execution: split array into chunks and assign chunks to processors (scheduling); generally more chunks leads to better load balancing (parallel slackness); order of execution must not influence the result (since order depends on scheduling), given by pure functions (no side effects, same result for same argument)
  - **reduce** (reduction)[28]: input: aggregate (x), binary associative operator ($\oplus$) with an identity I, output: $x_1 \oplus x_2 \oplus \ldots \oplus x_n$; result stays the same for sequential vs binary tree if operator is associative($(a+b)+c = a+(b+c)$); if operation is commutative ($a+b=b+a$), different scheduling is possible; e.g. sum, max
  - **prefix scan**: if it is an addition, it is a prefix sum; input: aggregate (x), binary associative operator ($\oplus$) with an identity I, output: ordered aggregate ($x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \ldots \oplus x_n$);
  - prefix[29] scan algorithm parallel version: addition example: 1st step is a reduction where two numbers are summed together and then pass their sum up the tree until it reaches the root i.e. bottom-up summing up all the values, two at a time; 2nd step is a down sweep where every node gets the

[26] important methods to be overridden: preStart(), onReceive()
[27] code in slides
[28] Remember the introductory lecture where we were asked how we could efficiently sum up all money in the lecture hall? There you go!
[29] If in node i, in_i is *not* added , the operation is called a *pre*-scan

sum of all the preceding leaf values passed whereas preceding is defined as pre-order[30]; **Have a look at slides 18 – 21** if in doubt

- application of pre-scan: line-of-sight, visible points (e.g. mountain tops) from a given observation point: point I is visible iff no other point between I and the observer has a greater vertical distance ($\theta_i = \arctan \frac{altititude_i - altitude_0}{i}$); compute angle for every point, do a max-pre-scan on angle array (e.g. 0,10,20,10,30,20 → 0,0,10,20,20,30), if $\theta_i > maxprevangle_i$ then $visible_i = true$ else $visible_i = false$; parallelizable parts: for loop to compute angles, for loop to compute visibility can be written as parfors (parallel for loops)
- **parfor**: iterations *can* be performed in parallel, work partitioning -> partition iteration space; potential source of bugs if thought of as a sequential loop (data races; think factorial)

Data Parallel Programming in Java 8

- Functional programming crash course: functions are first-class values (composition), pure functions (immutability); such function are called lambdas or anonymous functions
- Functions as values: functions can be passed to other functions as arguments (such functions accepting such arguments are called high-order functions), e.g. $map(f, list): f$, $filter(fn, list): f$
- Lambdas make programming more convenient
- Data parallel programming in Java 8 is done using streams, providing means to manipulate data in a declarative way, allowing for transparent parallel processing;
- Menu example: input: stream, output: stream, map/filter/etc. are applied; collect in the end, doesn't create a stream; overall translates a stream into a collection[31]
- Parallel streams: created by applying .parallel() on a stream, splits it up into chunks for different threads; implemented using ForkJoin

Exercise 5

- **Dining Philosophers**: easy deadlock if every philosopher acquires left fork; cyclic dependency needs to be broken using e.g. lock ordering; to get maximum number of philosophers eating, bundle forks (maximum for five philosophers is two)
- **Banking System**: wrapping every method in "synchronized" will lock the whole system and thereby losing the parallelism (plus a bottleneck is created where many threads try to acquire a lock); Java's intrinsic locks are reentrant[32] (= same thread can acquire same lock multiple times), thus transfer(a, a, x) is not a problem; to prevent deadlocks, lock ordering can be used (e.g. account id); to get a correct sum (incorrect during transactions), lock all accounts before getting its balance and not releasing the lock until *all* accounts are summed up (aka two-phase-locking; one phase were all locks are acquired and no locks are released and another phase where all locks are released and no locks are acquired), do not forget lock ordering; summing up can easily be parallelized since a sum is associative (and more, actually)

Exercise 6

- *look at the code!*
- Hints/takeaways: static variables are not associated with an object, they are per-class and can be used as such (helpful for the boat); R/W lock: the last reader should notify waiting writers, when trying to acquire a writer lock, first wait for (if applicable) the writer to finish writing and then wait for all readers to finish

[30] Depth-first, left-to-right
[31] Think SQL
[32] And of course also explicit locks using ReentrantLock

reading, when releasing the writer lock, notify all waiting threads on the end, save current thread to check for release-without-acquire[33]

Exercise 7

MISSING since official solution is yet to be published

Exercise 8

Please have a look at the assignment yourself

Sources

- Lecture slides from 252-0024-00L held at ETH during the spring semester 2014 by Prof. Dr. O. Hilliges and Dr. K. Kourtis, available at http://ait.inf.ethz.ch/teaching/courses/2014-SS-Parallel-Programming/
- Wikipedia (rarely)

Further Reading

- Conditions: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html

Bonus

You *implemented* .run() in a thread and you *call* .start()![34]

[33] Thread.currentThread()
[34] This might seem silly since it was mentioned quite a few times, but then again, some people…