Midterm I

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

- **Notes/comments/remarks on exercises 3&4**

The missing parts will most likely be added Thursday, March 20, 2014[1]. The document's version will then change from 1.1b to 1.2b

Table of Contents

Info

There is no claim for completeness. All warranties are disclaimed.

[1] I'm in a Thursday exercise group

Study Part

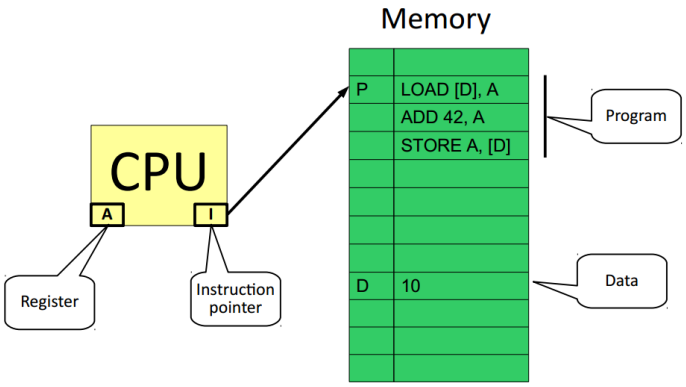
The contents are structured according to the lecture slides. Some of the texts are 1:1 copies from the lecture slides available at http://ait.inf.ethz.ch/teaching/courses/2014-SS-Parallel-Programming/. For the sake of readability and quicker typing, those excerpts are often simply in double quotes. And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

01 Course Overview

- Even though Moore's Law[2] is still valid, heat and power consumption are of primary concern. These challenges can be overcome with smaller and more efficient processors or simply more processors. TO make better use of the added computation power, parallelism is used.
- Parallel vs. concurrent: (quoted); in both cases, one of the difficulties is to actually determine which processes can overlap and which can't
  - Concurrent: focus on which activities may be executed at the same time (= overlapping execution)
  - Parallel: overlapping execution on a real system with constraints imposed by the execution platform
- Parallel/concurrent vs. distributed: In addition to parallelism/concurrency, systems can actually be physically distributed (e.g. BOINC).
- Concerns in PP: (quoted)
  - Expressing parallelism
  - Managing state (data)
  - Controlling/coordinating: parallel tasks and data

02 Parallel Architectures

- Turing machine: (quoted) infinite tape, head for r/w, state register
- Computers today: consist of CPU, memory and I/O; "stored program" i.e. program instructions are stored in memory and program data, too (von Neumann)
- Since access memory became slower than accessing CPU registers, CPUs now have caches which are closer (and thus faster but also smaller) to the CPU (locality of reference → principle of locality); L1, L2, L3, … cache

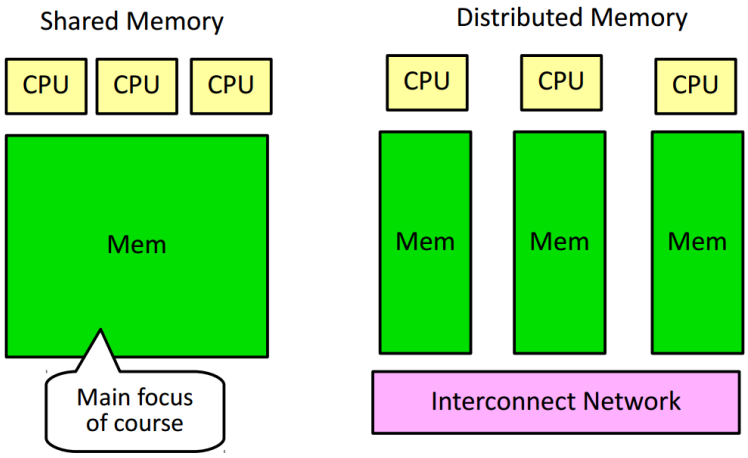


- "Applying parallelism to improve sequential processor performance: vectorization[3], pipelining[4], Instruction Level Parallelism (ILP)"
- Pipelining: There is a lead in and a lead out where system is warming up/cooling down (resp.) and full utilization (which is to be maximized). On a CPU:

- ILP: "Pipelining; Superscalar CPUs: Multiple instructions per cycle / multiple functional units; Out-of-Order (OoO) execution: Potentially change execution order of instructions, as long as the programmer observes the sequential program order; Speculative execution: predict results to continue execution"

[2] Actually an observation; "The number of transistors on integrated circuits doubles approximately every two years"
[3] Applying an operation on every element of a vector in parallel instead of sequentially; instead of 1-at-a-taime, N-at-a-time
[4] Think laundry

- For a long time, Moore's Law and ILP made sequential programs exponentially faster but due to power dissipation (expensive to cool CPUs), CPUs becoming faster than memory access and not being able to ILP a program anymore, it was "no longer affordable to increase sequential CPU performance". The solution was multicore processors which, however, first needs programmers to write programs which can actually take advantage of the new hardware.



- "Shared memory architectures: SMT (Intel's Hyperthreading; Simultaneous MultiThreading), Multicores, SMP, NUMA"
- SMT: "single core, multiple threads, ILP vs. multicore: ILP multiple units for one thread, SMT multiple units for multiple threads"
- CMP/multicores: "dual, quad, x8 etc. ; each has its own hardware, yet might share part of the cache hierarchy"
- SMP: "multiple chips (CPUs) on the same system: CPUs share memory – same cost to access memory; CPU caches coordinate – cache coherence protocol"
- NUMA/Non-uniform memory access: memory is distributed (local/remote) at the cost of speed, shared memory interface
- Distributed memory[5]: organized in clusters
- Flynn's taxonomy: [S|M]I[S|M]D where S = single, M =multi, I = instruction, D = data; used to classify different types of architectures
- GPUs are badass! (and they're great for data parallelism)

03 Basic Concepts

- Performance in sequential execution: computational complexity O, Θ and execution time
- Sequential programs are much easier to write, yet parallel programming has better performance
- Parallel performance formulae:

  $T_1: sequential\ execution\ time, \quad T_p: execution\ time\ on\ p\ CPUs$

  $T_p > \frac{T_1}{p}, performance\ loss, normal;\ T_p = \frac{T_1}{p}, perfection;\ T_p < \frac{T_1}{p}, sorcery$

  $S_p: speedup\ on\ p\ CPUs;\ S_p = \frac{T_1}{T_p}$

  $S_p = p, linear\ speedup, perfection;\ S_p < p, sublinear\ speedup, performance\ loss$

  $S_p > p, superlinear\ speedup, sorcery$

  $Efficiency: \frac{S_p}{p}$

  $Amdahl's\ Law$[6] $(b = sequential\ part, 1 - b = parallel\ part)$:

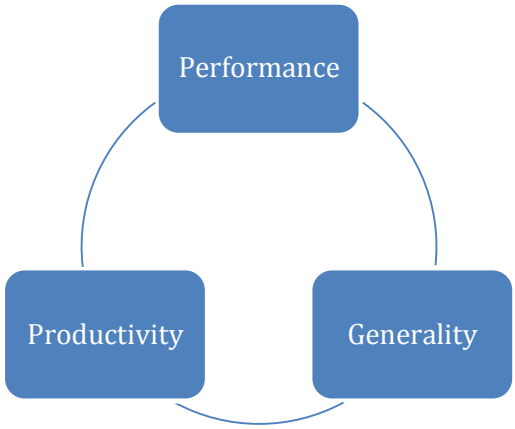
  $T_p = T_1 \cdot \left(b + \frac{1-b}{p}\right), S_p = \frac{p}{1 + b \cdot (p-1)}$

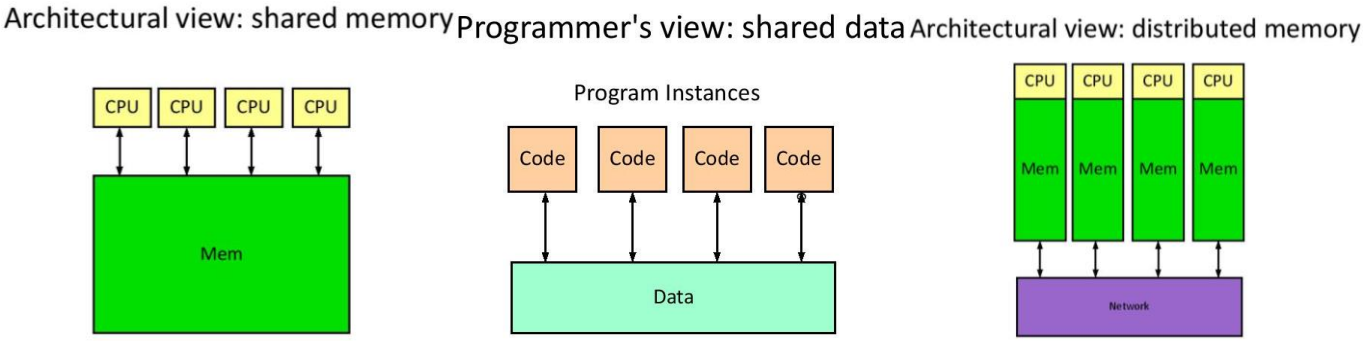
[5] See Top500

[6] Very optimistic approach, Gustafson was more realistic: it considers problem size, runtime (and not problem size) is constant, more process can solve larger problems in the same time, parallel part scales with he problem size

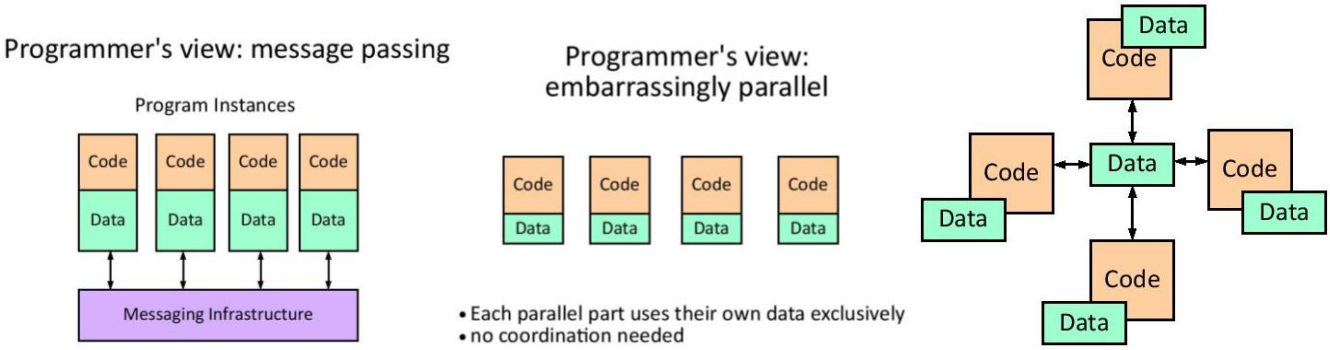
Gustafson's Law ($b = sequential\ part$): $T_1 = p(1-b)\cdot T_p + b\cdot T_p, S_p = p - b(p-1)$

- Scalability: how well a system reacts to increased load; in PP: speedup with more processors (even to ∞), linear speedup is desirable
- Reasons for performance loss: program may not contain enough parallelism, overhead (due to pp), architectural limitations (think group work/presentation)
- Concurrency vs. parallelism: "Concurrency is: a programming model, programming via independently executing tasks, about structuring a program, example: network server, a concurrent program does not have to be parallel; Parallelism is about execution, concurrent programming is suitable for parallelism"
- Code and data – code doesn't change over time while data does





Programmer's view: non-shared & shared data



- Expressing parallelism: Work partitioning (splitting up work of a single program into parallel tasks) which can be done manually (task parallelism; user explicitly expresses tasks) or automatically by the system (data parallelism; user expresses and operation and the system) (quoted)
- Work partitioning & scheduling (quoted): work partitioning: split upwork into parallel tasks, (done by user or system), a task is a unit of work, also called: task decomposition; scheduling: assign tasks to processors, (typically done by the system), goal: full utilization (no processor is ever idle)
- Coarse vs. fine [task] granularity: fine granularity is more portable and better for scheduling, parallel slackness[7], but overhead may grow (too) big

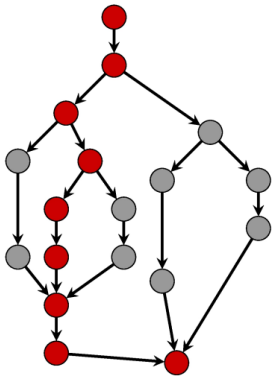
[7] expressed parallelism ≫ machine parallelism

- Coordinating tasks: enforcing a certain order since e.g. task X needs the result of/has to wait for task A to finish; example primitives: barrier, send(), receive()
- Managing state concerns: shared vs. distributed memory architectures; which parallel task access which data and how (r/w); potentially split up data; task, then data or data, then tasks (quoted)
- Coordinating data access (quoted): distributed data: no coordination (e.g., embarrassingly parallel), messages; shared data: controlling concurrent access, concurrent access may cause inconsistencies, mutual exclusion to ensure data consistency

04 Parallel Models

- PP is not uniform, many different approaches (!)
- PP paradigms (quoted): task parallel: Task Parallel: Programmer explicitly defines parallel tasks (generic, not always productive); Data parallel: An operation is applied simultaneously to an aggregate of individual items (e.g., arrays) (productive, not general)

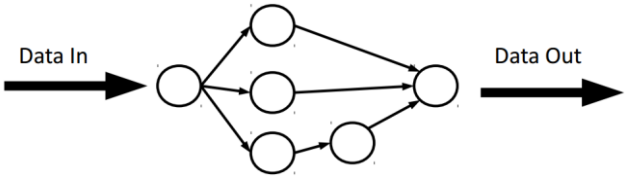
Task Parallelism



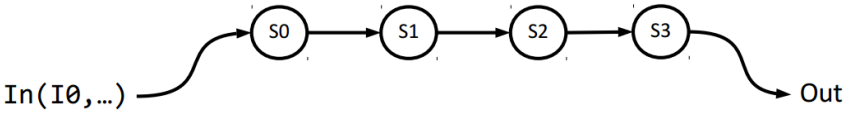
- Tasks execute code, spawn other task and wait for results from other tasks
- Tasks *can* execute in parallel (decided by the scheduler), task graph is dynamic (unfolds) – wider task graph = more parallelism
- T_1: $total\ work\ (time\ for\ sequential)$
  $\frac{T_1}{T_p}$: $speedup, lower\ bounds\ T_p \geq \frac{T_1}{p}, T_p \geq T_\infty$
  T_∞: $span, critical\ path, longest\ path$
- Scheduling: assigns task to processor, upper bound $T_p \leq \frac{T_1}{p} + T_\infty$ can be achieved with a greedy scheduler (all processors are executing tasks, if enough tasks available), optimal with a factor of 2, linear speedup for $\frac{T_1}{T_\infty} \geq p$
- Work stealing scheduler: provably $T_p = \frac{T_1}{p} + O(T_\infty)$, empirically $T_p \approx \frac{T_1}{p} + T_\infty$, linear speedup if $\frac{T_1}{T_\infty} \gg p$, parallel slackness (granularity)
- Common structure for divide & conquer (e.g. accumulator/$\sum a_i$):

```
Divide and Conquer:
    if cannot divide:
        return unitary solution (stop recursion)
    divide problem in two
    solve first (recursively)
    solve second (recursively)
    combine solutions
    return result
```

- Task graph can also be static, e.g. pipeline, streaming, dataflow
- Dataflow: Programmer defines: what each task does and how the tasks are connected



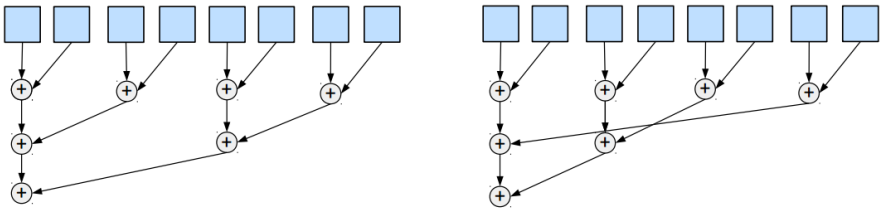
- Pipeline[8]: time unit is determined by the slower/slowest stage (→stalling), every stage should take the same amount of time; can be achieved by using splits and joins for parallel stages



- Scheduling dataflow programs: scheduling: assigning nodes (tasks) into processors, $n < p$: cannot utilize all processors,$n = p$: one node per processor, $n > p$: need to combine tasks; portability, flexibility (parallel slackness), balancing, minimize communication (graph partitioning); dataflow is a good match for pp, since the programmer isn't concerned with low-level/edge implementation details; can be generalized with feedback loops (performance becomes more difficult, though)

Data Parallelism

- In data parallelism, the programmer describes an operation on an aggregate of data items (e.g., array); work partitioning is done by the system; declarative: programmer describes what, not how
- Map/reduce: map example: $B = 2 \cdot A$ where actually $b_i = 2 \cdot a_i \forall i$; reduce example: d&q accumulator making use of associativity and commutativity (second example: max()[9])



- Parallel loops can be used for work partitioning by adding generality yet possibly introducing "weird" bugs due to data races (e.g. operation on a_i depends on a_{i-1})

Managing State

- Main challenge for parallel programs

IMMUTABILITY	ISOLTEAD MUTABILIY	MUTABLE/SHARED DATA
– data do not change – best option, should be used when possible	– data can change, but only one execution context can access them – message passing for coordination – State is not shared – each task holds its own state – (async) messages – Models: actors, CSP (Communicating Sequential Processes)	– data can change / all execution contexts can potentially access them – enabled in shared memory architectures – however: concurrent accesses may lead to – inconsistencies – Solution: protect state by allowing only one execution context to access it (exclusively) at a time; e.g. using locks (good performance, correctness issues) or transactional memory (correct, bad performance)

[8] Already discussed earlier to some extent
[9] Similar: prefix scan

05 Introduction to Programming

I'm not going to talk about syntax and the like, you can read up on this in the lecture slides.[10]

- Old Egyptian Multiplication: say you want to multiply a with b. In one column you keep writing down 2^n as long as this is $\leq a$. In the other column you start with b (for row a=1) and then keep doubling the last row until you reach the last row of 2^i. Then you figure out which 2^i form a, cross out the other rows and then add up the corresponding rows. Example: $27 \cdot 12 = (16 + 8 + 2 + 1) \cdot (12 + 24 + 96 + 192) = 324$

27 = 16+8+2+1	**12**
1	12
2	24
~~4~~	~~48~~
8	96
16	192
~~32~~	~~384~~

- Russian Peasant Multiplication: In one column (division column) you keep dividing the number a (while floor()[11]-ing if need be)until you reach 1 while in the other column (multiplication column) you keep multiplying b as long as the corresponding row in the division row hasn't yet reached 1. In binary you keep deleting the LSB in the division column while adding 0s in the multiplication column. Then you cross out lines with an even number in the division column and sum up the values in the remaining multiplication column. This method is super great for CPUs! **Note:** If the multiplicand is odd, you have to add a in the end (die to underestimating a)

  Formally: $a \cdot b \begin{cases} a, if\ b = 1 \\ 2a \cdot \frac{b}{2}, if\ b\ even \\ a + \left(2a \cdot \frac{b-1}{2}\right), else \end{cases}$, recursive: $f(a,b) = \begin{cases} a, if\ b = 1 \\ f\left(2a, \frac{b}{2}\right), if\ b\ even \\ a + f\left(2a, \frac{b-1}{2}\right), else \end{cases}$

- Important concept of Exception Handling; main keywords: try{…}catch(…){…} and throw[s] …

06 Java Basics

- Java is an interpreted (using compiled byte code) language running in the Java Virtual Machine (JVM), making it possible to run on virtually any computing device
- When writing an algorithm: KISS (keep it simple and stupid), group it logically, try to write re-usable code, DRY (don't repeat yourself)
- In Java, the 'main' method has a special significance, it gets called at runtime automatically as an entry point
- Java uses types (strongly typed) – primitive[12] (byte, short, int, long, float, double, char, boolean) and object (String, and all the rest); everything has a type (and needs to be declared as such); Types can be cast using myInt = (int) myFloat

07 Loops – Objects – Classes

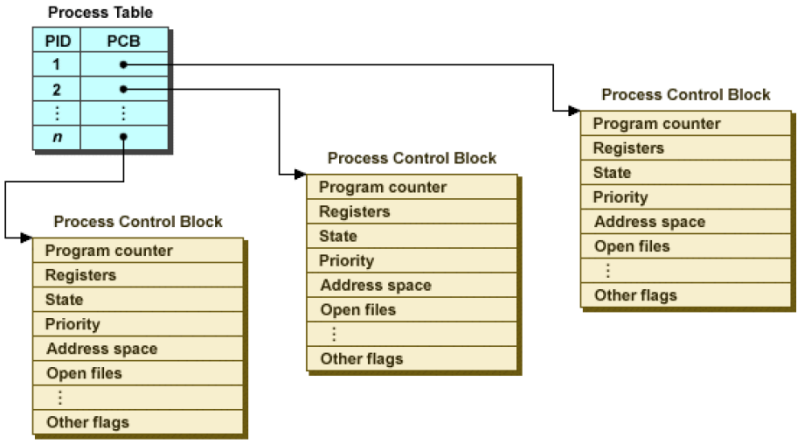
- Loops can be definite (think 'for'), indefinite (think 'while') and sentinel[13] (until a sentinel value is seen)
- Fencepost problem (off-by-one error): (Wikipedia) "It often occurs in computer programming when an iterative loop iterates one time too many or too few.", can be avoided by e.g. using a do-while loop

[10] No offense!
[11] Round down to the nearest integer, $\lceil n \rceil$
[12] Which are not real objects, instead they have a wrapper class
[13] German (here): Markierung

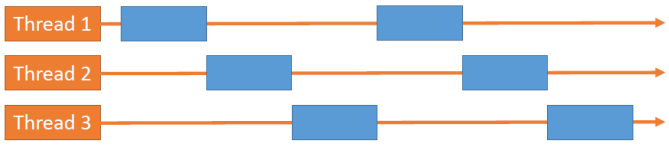
- Arrays[14] are zero-based and use key-value pairings (or index-value), play well with for(each) loops; arrays are reference types; Arrays can throw ArrayIndexOutOfBoundsException if not implemented correctly/thought through
- Strings aren't created with 'new', they actually are a 'char'-array; strings can't be compared with '=='[15] (reference comparison of objects)
- Classes are code (just like a blueprint) while objects are instantiated classes (code vs. runtime); objects contain data (variables) and objects (methods); classes are (often) abstractions
- Null is special (can be used for a non-argument, return value for failed calls, default value of a variable etc.)
- Encapsulation: very important in OO – every object has internal and external view, it's also a form of protect (information hiding), methods maintain data integrity, different visibility keywords (public (everywhere), private (only from this class), protected (current package and subclasses, regardless of package)); benefits: protects from unwanted access, implementation can be changed later, object's state can be constrained (invariants)
- Java uses packages (for namespacing)
- 'this' refers to the implicit parameter inside your class
- Class methods are marked with 'static' (can be called from a static context, e.g. main()); they're often generic and need no access to object variables and methods; serve as utility functions

08 Threads

- Multitasking: concurrent execution of multiple tasks: time multiplexing on CPU (creates impression of parallelism even on single core system); allows for asynchronous I/O
- Process context: instruction counter, register content, variable values, stack content, resourcing
- Process states: main memory: created, waiting, running, blocked, terminated; page file/swap space: swapped out waiting, swapped out blocked
- Process management: CPU time, memory; tasks managed by OS: start/terminate processes, control resource usage, schedule CPU time, synchronization of processes, inter-process communications
- Process control blocks (PCB)(see image) – process level parallelism can be complex and expensive



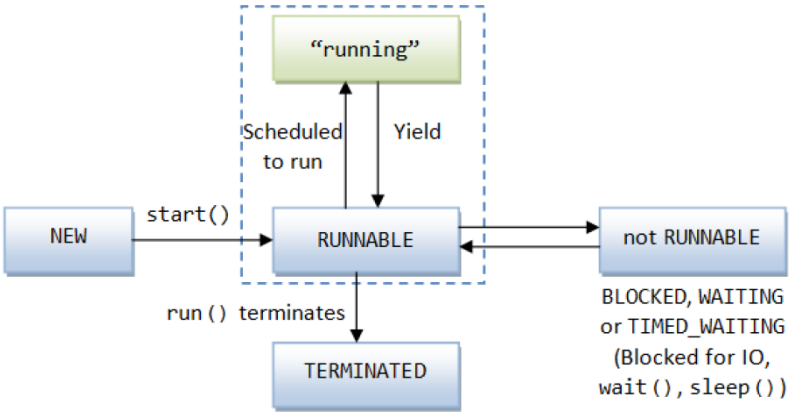
Multiple threads sharing a single CPU



Multiple threads on multiple CPUs

[14] 'int diaryEntriesPerMonth[] = new int [31]'
[15] 'equals()'

- Threads are light weight processes, they are independent sequences of execution but multiple threads share the same address space, they aren't shielded from each other but share resources and can communicate more easily, context switching is much more efficient; advantages: reactive system by constant monitoring, more responsive to user input (GUI interruption), server can handle multiple clients simultaneously, can take advantage of parallel processing



- Overriding methods: a subclass' method replaces a superclass' version of the same method
- Interface: list of method a class can implement; gives you an is-a relationship and without code-sharing (inheritance shares code)
- Creating threads in Java: extends 'java.lang.Thread' (override method, run()/start(); implement 'java.lang.Runnable'[16] (run()), if already inheriting from a class[17], 'Thread' implements 'Runnable'
- Every Java program has at least one execution thread (which calls main()), each call to Thread.start() creates a new thread (but not just the creation of a Thread object and run() doesn't start a thread either), program ends when all threads finish yet they can continue even if main() returns
- A thread has the following attributes (getters and setters): ID, name, priority (1…10), status (new, runnable, blocked, waiting, time waiting, terminated)
- A thread can throw 'InterruptedException'; can be requested by Thread.interrup() but can be ignored; fain grained control with isInterrupted(), interrupted()
- Checked exceptions (quoted): represent invalid conditions in areas outside the immediate control of the program (network outages, absent files); are subclasses of Exception; a method is obliged to establish a policy for all checked exceptions thrown by its implementation (either pass the checked exception further up the stack, or handle)
- Unchecked exceptions (quoted): represent conditions that, generally speaking, reflect errors in your
- program's logic and cannot be reasonably recovered from at run time (bugs); subclasses of RuntimeException, and are usually implemented using IllegalArgumentException, NullPointerException, or IllegalStateException; a method is not obliged to establish a policy for the unchecked exceptions thrown by its implementation (and they almost always do not do so)
- (quoted) Threads can make concurrent (and asynchronous) workflows faster even on single core machines. If execution units are well separated they can make programs even simpler to write.; for data heavy and compute intensive parallel programs there is usually no speed-up beyond the number of physical cores; Even then scheduling and communication overhead might reduce performance gains

09 Synchronization: Introduction to locks

- need for synchronization: races
- in a sequential program, 1:1 of program and data, in a parallel (multi thread) program, many different threads need to access data - data needs to be protected since concurrent access *might* (-> in sequential

[16] 'public class [Name] implements Runnable {}'
[17] Java doenst have multiple inheritance

program, bugs become apparent quickly (Exception: boundary conditions)) lead to inconsistencies, concurrent access bugs often depend on execution conditions (#CPUs, load, timing) and (thus) they're difficult to reproduce
- sequential algorithms assume they act alone, thus them acting on data is unsafe; hardware/software optimizations assume sequential execution (which can mess up things)
- Example: circular doubly linked list: every node has a forward and a backward pointer and the list is circular (last and first are linked); easy to insert sung four operations - remove is two operations (easy as well); remove() with 2 threads can create a mess (depending on the scheduling)
- race condition: correctness depends on relative timing; data race: unsynchronized access to shared mutable data; most race conditions are due to data races (but not always)
- avoiding race conditions: very difficult to consider all possible execution interleavings; instead use locks, locks - atomicity via mutual exclusions
- atomicity: operations A and B are atomic with respect to each other, if "thread 0 executes all of A, thread 1 executes all of B and Thread 0 **always** perceives either all of B or none of it"; intermediate state cannot be observed, only start and end
- atomic mutual exclusion works on sequential algorithms (with no or little adaption)
- thread safety: apply OO techniques (encapsulation): design "thread safe classes" which encapsulate any needed synchronization so that the clients don't need to worry about that; you should build your program by composing thread-safe classes (which isn't always easy); definition: "Behaves correctly when accessed by multiple threads" and doesn't require synchronization from users
- concurrent access breaks many assumptions from sequential programming (counter-intuitive); performance vs productivity
- we cannot make any assumptions about relative execution Speed of threads (-> synchronization with sleep() is wrong); even a single instruction is not atomic (value++ needs READ, INCREMENT, STORE); can be avoided using Java keyword "synchronized"
- Basic synchronization rule: Access to shared and mutable state needs to be **always** protected - Access includes both reads and writes (fine print: you can break it if you know and understand architectural details)
- Locks: a lock object instance defines a critical section; lock->enter, unlock->leave; there can be only one (thread)
- Java has intrinsic locks, each Java object contains a lock (built-in, for your convenience yet undesired implications (size)); can be used via they keyword "synchronized" on code blocks or entire function
- reentrant lock: (example: two synchronized functions, and one function calls the other) if a thread tries to acquire a lock it already holds it succeeds (normally this leads to deadlock, where the program is unable to proceed); in Java intrinsic locks are reentrant
- reentrant: per-thread acquisition, non-reentrant: per-invocation locks (some argue this is better since you need to think harder); trade-off: flexibility <-> productivity
- explicit locks: not a replacement for "synchronized", provides more flexibility (additional calls, non-block structured locks)
- synchronized vs function call: synchronized: part of the language, less error-prone but less flexible; function call: library, error-prone but more flexible
- reentrant lock using lock interface with try/catch/finally{unlock}
- Example: synchronizing circular doubly linked list: if every single call has its own "synchronized", interleavings are still possible, correctness depends on the semantics of the program, operations need to be synchronized *properly* (one of the reasons why PP is hard)
- rule: to preserve state consistency, update related state variables in a single atomic operation (simple approach: use "synchronized" on (every) method)

- Java servlets (implementing an interface), multiple threads for better performance are a good idea as long as they're thread-safe, they **have to**
- counters can be "synchronized", too, for thread-safety
- use built-in mechanisms where available (e.g. java.util.concurrent.atomic.AtomicLong) - caching results (memorization): cache input->result; might save expensive computation (followed by storing the result in the cache using .clone()); yet one might lose parallelism by using "synchronized" on the whole function -> use "synchronized" only on critical sections (for cache r/w access) instead of on the whole function
- locks and Amdahl's Law: at some point (if you keep adding processors), the "synchronized" parts will dominate (so keep them at minimum)

10 Synchronization: Using Locks and Building Thread-Safe Classes

- - model: n threads, m resources
- coarse-grained locks (big lock), locking all resources (critical section), threads are serialized, but bad performance
- fine-grained locks, e.g. every resource is protected by one lock (note: 1 thread may have multiple locks), better performance; problems: using multiple locks for one thread, deadlock is possible
- deadlock: no thread is able to continue, caused by circular dependencies; runtime condition; necessary conditions: mutual exclusion (at least one resource must be non-shareable), had and wait (a thread holds at least a lock that it has already acquired, while waiting for another lock), no forced lock release (locks can only be released voluntarily by the threads (and not by system)), circular wait (e.g. p[1] waits for p[2] and so on and p[n] waits for p[1]); mutual exclusion, hold and wait and no forced lock release can't be broken with "synchronized" and breaking them leads to complicate synchronization schemes
- breaking the circular wait condition: set of global order of locks, acquire locks respecting that order, impossible to create a condition of circular wait -> impossible to deadlock
- Example: Hash Table (key, hash function, buckets, collision lists); attempt to make it thread-safe: huge lock around the whole data structure (apply hash() to key, locate bucket, search for key in the list and, if found, return value) - bad, Amdahl's Law: synchronized part will dominate eventually; per-bucket lock (search for key in the list and, if found, return value)
- Example: Hash Table: per-bucket locks - discussion: operations involving only a single key (insert, lookup, remove) are easy; swap(key1, key2): exchanges values between those two keys *atomically*, locking with a single lock is trivial, but what about multiple locks?: 1st attempt: lookup() twice, then insert() twice, each one locked: intermediate state might be observed between the two insert()s, 2nd attempt: lock both buckets -> no intermediate state can be observed (locate both buckets and then, while synchronized, search for both keys and swap them), however there's the possibility of a deadlock which can be avoided with lock ordering (using the hash table's indexes; special case for the same bucket if locks aren't reentrant)
- Visibility – synchronization also enforces visibility
- in example (42) the reason(s) for the problem are: until a few years ago, sequential performance was the main focus, thus optimizations were done with respect to sequential programs (CPU/compiler tuning, out-of-order execution) are only guaranteed for sequential execution (and cancelling them isn't affordable) and thus counter-intuitive behavior in parallel settings is common
- building thread-safe classes: immutable and stateless are always thread-safe (assuming correct Initialization), mutable and shared data needs to be protected; try to avoid synchronization since it's difficult to reason about locks and there are performance issues (->immutable/stateless)
- 'final' keyword in Java: final class cannot be subclassed, final method cannot be overridden or hidden, final variable can only be initialized once (and only in the class constructor) (but referenced object *can* be changed)

- immutable objects in Java - properties: 1) its state cannot be modifier *after* construction, 2) all fields are final, 3) it's *properly constructed* (= final fields need to be set in the object constructor; when the constructor is done, the object is immutable; while the constructor runs, the object is mutable; it should not be access during that time); they are always thread-safe!
- Generics – motivation: the implementation of e.g. a linked list is independent from the indulging element used (implementor perspective); static checks for linked lists operations (user perspective); decoupling of data structure and algorithms that operate on them
- improper construction: 'this' escapes
- 'AtomicReference<V>' is an atomic access to a reference of V (comparable (with care) with 'volatile')
- immutable vs non-immutable objects: immutable objects are special which is specified in the Java memory model and they don't require safe publication while non-immutable objects need to published safely since no ordering guarantees are proved by the memory model and a thread might observe a non-safely published object in an inconsistent state

Exercises 1

- Nothing of particular interest

Exercises 2

- Keep Amdahl's Law in mind
- Know how to draw/analyze/optimize a pipeline and calculate speedup etc.
- Height of a tree is $\log_2 n$ [18]
- Loop: if an element's operation depends on the value of a previous element, you can't parallelize that loop. If an element's operation depends only on its own value, you can parallelize that loop.

Exercise 3

- **MISSING**

Exercise 4

- **MISSING**

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)

[18] Not always, but if you're asked to find some formula, keep log in mind