

Parallele Programmierung

1. Producer Consumer
2. Java Virtual Machine
3. Category of loops
4. Garbage collector
5. Bestandteile von JVM
6. Parallelism on CPU
7. Java Threads
8. Synchronized Blocks
9. Implizit vs. expliziter Parallelismus
10. Approaches to introduce parallelism to single core CPH
11. Parallel performance
12. Amdahl's Law
13. Gustafson's Law
14. Work partition and scheduling
15. Executor Service
16. Fork Join Pool
17. Rekursive Berechnung
18. Prefix-Sum-Problem
19. Locks
20. Race Conditions
21. Java Volatile
22. Peterson's Lock
23. Fiber Lock
24. Bakery Algorithmus
25. Atomic Operations
26. Java atomic
27. SpinLock
28. Semaphores
29. Barriers
30. Problems with Locks
31. Monitors (-> synchronized Blocks)
32. Mehrere Condition auf einer Lock
33. Readers-Writers Lock
34. Coarse-Grained-Locking
35. Fine-Grained-Locking
36. Optimistic Locking
37. Lazy Locking
38. Lazy SkipLists
39. Progress conditions
40. ABA Problem with compareAndSet
41. Lock-Free
42. Linearisierbarkeit
43. Sequential Konsistenz
44. Consens Protocol / Number
45. Transactional Memory
46. Distributed Memory & Message passing
47. Sorting Networks
48. Preburger Sets & Relations
49. Scheduler

Producer-Consumer

- Producer produziert zu verarbeitende Dinge
- Consumer verarbeitet Dinge
- Datenstruktur: Queue, Stack, Circular Queue, usw.
- Implementationen
 - Ohne synchronized: Starvation, endlos warten
 - Mit synchronized, ohne while: Kein recheck, if consumable
 - Mit synchronized und while: while (!consumable) wait()

Java Virtual Machine (=JVM)

- Interprets virtual CPU → can run on any system
- Source Code → Bytecode → Architecture Code

Compiler JVM
- High-Level-Code: Must be translated (Java)
- Low-Level-Code: Native (= C)

Category of loops

- Bounded-Loops: Man weiß genau, wie viele Iterationen
- Finite-unbounded-Loops: z.B. Wegen User-Input / random
- Infinite-unbounded-Loops: while(true) (= z.B. Webserver)

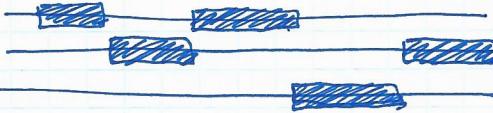
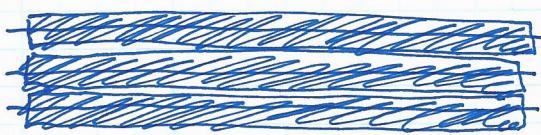
Garbage Collector

- Je mehr threads, desto langsamer
1. Stops all threads (um interleaving zu vermeiden)
 2. Mark all reachable objects that have a active reference
 3. Frees/removes non reachable objects
 4. Releases all threads

Bestandteile von JVM

- Memory allocation: Triggered by "new"
- Java-to-OS: Mapping Java features to OS
- Interpreter: Java-Code → Bytecode
- Native-Interface:
 - Kein Java-Programm läuft komplett mit Java
 - Gibt Programmteile mit Interface an C/C++ weiter nach unten
- Verification: Type checking

Parallelism on CPU

- One-Core-CPU: Thread 1 
Thread 2
Thread 3
- Multi-Core-CPU: Thread 1 
Thread 2
Thread 3

Java-Threads

- Methods - State

- start: starts new thread, that calls run / callable
- sleep (time): thread sleeps and holds lock for time }
 ↳ Blocked
- wait: thread waits and releases lock for notify }
 ↳ Only in synchronized
- notify: notifies (all) threads to wake up }
- join: waits for thread to die

- Create new thread

- Instantiate a subclass of java.lang.Thread class

↳ Thread t = new Thread () { public void run() {

 // code }

};

t.start();

- Implement Runnable into Class

↳ public class MyThread implements Runnable {

| public void run() { }

| }

↳ MyThread op = new MyThread();

Thread t = new Thread(op);

t.start();

- Extend Thread-Class

↳ public class MyThread extends Thread {

| public void run() { }

| }

↳ Thread t = new MyThread();

t.start();

- Thread-Attributes

- Thread t = Thread.currentThread();

- t.getId() (= gets current ID)

- t.setName("n") (= sets name)

- t.setPriority(1-10) (= sets thread priority)

- t.getState() (= current state of thread)

- t.isAlive

- Dead state

- When it's finished with run() execution

- When it is destroyed with thread.destroy()

- Object reference

- synchronized: called on any object

- wait/notify: called on any object inside synchronized

- join/start: called on any thread

} Monitor

Synchronized Blocks

- Enforces mutual exclusion lock on any object
 - synchronized (object) {}
 - A thread can request to lock an object it has already locked (nested locking)
 - The more granularity, the more performance
 - ↳ Wenn ich ganze array locke, muss ich warten, obwohl die threads verschiedene indexe benutzen
 - Synchronized ist pessimistisch, auch wenn der andere thread die Variable nicht benötigt, locke ich sie
 - wait(); (=releases object lock, thread waits priority queue)
 - notify(); (=wakes the highest-priority-thread)
 - notifyAll(); (=wakes all waiting threads)
 - You only can synchronize objects, not primitive types
 - Exception in synchronized
 1. Method is locked
 2. Exception is thrown
 3. JVM looks up exception table
 4. release lock

↳ If an exception is thrown during the synchronized block, the compiler generates an exception table to safely free the locked object

sagt, wenn ich an Zeile X exception habe, dass ich dann zu Ort Y muss
 - Synchronized can be deadlocked
- T1:

 - ① synchronized(A)
 - ② synchronized(B)

T2:

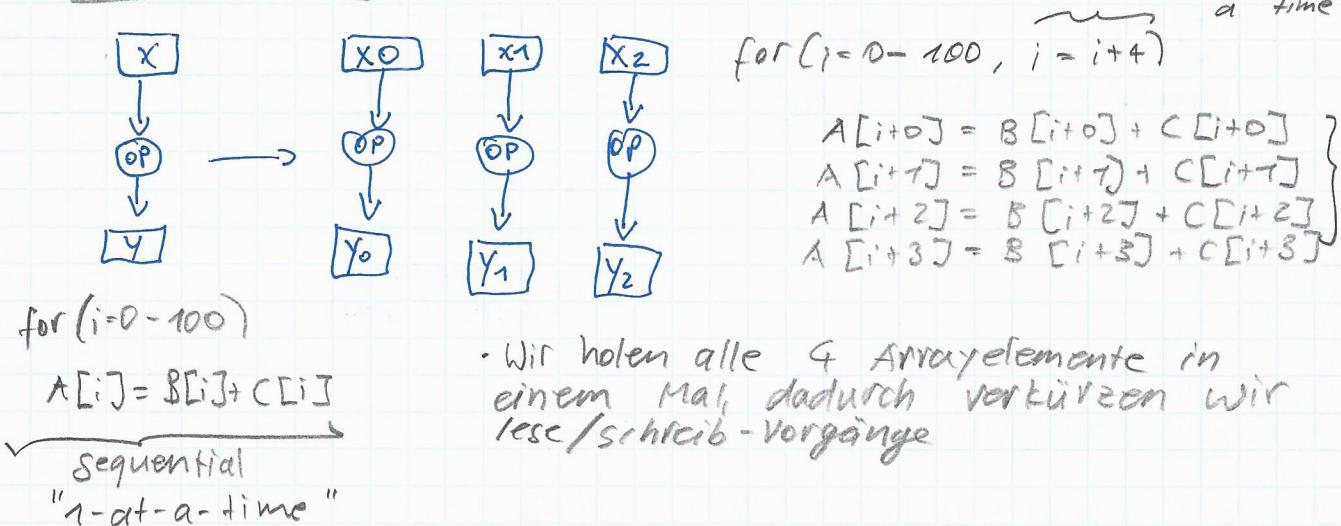
 - ② synchronized(B)
 - ③ synchronized(A)
- Nested Blocking
Thread 1 is holding lock A and waits for signal from Thread 2.
Thread 2 needs the lock A to send the signal

Implizit vs. expliziten Parallelismus

- Implizit: Guter Compiler transformiert sequentielles Programm (z.B. Loop) in parallele Version
- Explizit: Wenn ich selbst parallelen Code schreibe

Approaches to introduce parallelism to single-core CPU

Vectorization



• Wir holen alle 4 Arrayelemente in einem Mal, dadurch verkürzen wir Leser/Schreib-Vorgänge

Pipelining

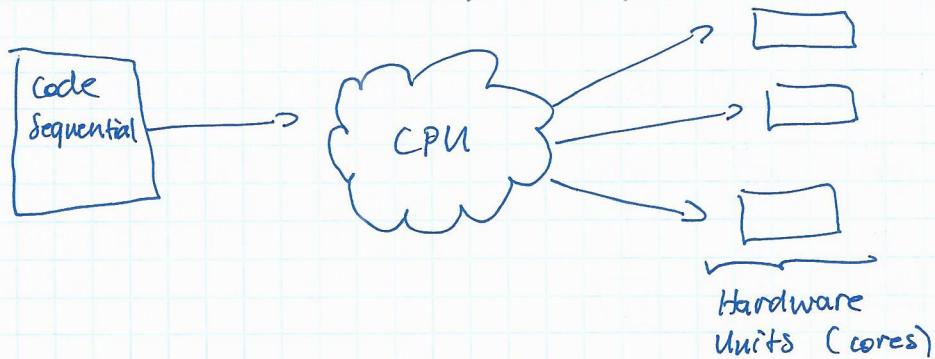
- Reusing the available computation-power immediately, when it's finished
- Man unterteilt Berechnungen in Stages
- Throughput:
 - Amount of work, that can be done in a given time "Amount"
 - CPH: Amount of instructions in a second
 - Pipeline-throughput approx.: $\frac{1}{\text{time of longest stage}}$
 - The higher the better
- Latency:
 - Time to perform the execution with all stages "Time"
 - Dauer eines ganzen Durchlaufs
 - Pipeline balanced: sum of all stages
 - Pipeline unbalanced: #stages · time of longest stage

- Balanced Pipeline:
 - Hard to accomplish
 - When all stages have same execution time
 - No stalling

- Unbalanced Pipeline:
 - Different execution time

Instruction Level Parallelism

- Sequentialles Programm
- CPU verteilt Anweisungen auf alle Cores



Parallel Performance

- T_1 := „How long a program takes, on 1 CPU“
- T_p := „How long a program takes, on P CPUs“
 - Ideal: $T_p = T_1/p$ (=Linear)
 - ↳ Jedoch geht immer ein wenig Performance verloren
 - ↳ sub-linear $\frac{T_1 \text{ parallel}}{P}$
- T_{so} := „Nur noch der rein sequentielle Teil bleibt über“
- S_p := „Parallel Speedup on P cores“ = T_1/T_p
- Efficiency := „Sagt aus, wie ausgelastet jede CPU ist“ = S_p/P
- Example

$$\begin{aligned}
 & \cdot \text{Sequential part: } 20\% \\
 & \cdot \text{Parallel part: } 80\% \\
 & \cdot T_1 = 10 \text{ sek}
 \end{aligned}
 \quad \left\{ \begin{array}{l} \cdot 2 \text{ sek} \\ \cdot 8 \text{ sek} \\ \hline 8 \end{array} \right\} \quad \left\{ \begin{array}{l} T_8 = 2+1=3 \text{ sek} \end{array} \right\}$$

Amadahl's Law (= Pessimistisch)

- Time serial work ($= W_{\text{ser}}$)
- Time parallel work ($= W_{\text{par}}$)
- T_{ser} gives us a bound on speed up, because we can't speed up serial-work

$$T_p \geq W_{\text{ser}} + \frac{W_{\text{par}}}{p}$$

$$S_p \leq \frac{W_{\text{ser}} + W_{\text{par}}}{W_{\text{ser}} + \frac{W_{\text{par}}}{p}} = \frac{T_1}{T_p}$$

- f is the serial fraction of the work, T_1 is given

$$W_{\text{ser}} = f \cdot T_1$$

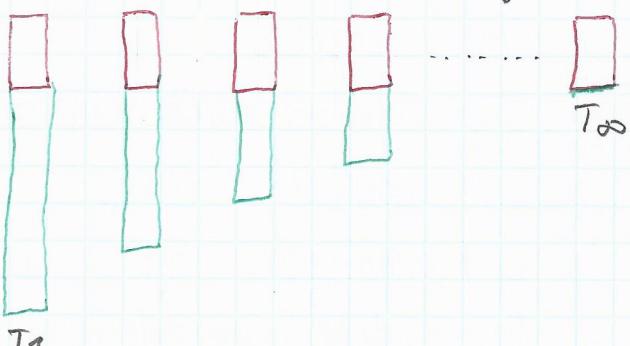
$$W_{\text{par}} = (1-f) \cdot T_1$$

$$S_p \leq \frac{1}{f + \frac{1-f}{p}}$$

$$S_p \leq \frac{1}{f}$$

$f = 0,2$

$W_{\text{ser}} = 20\%$
 $W_{\text{par}} = 80\%$

Gustafson's Law (= Optimistisch)

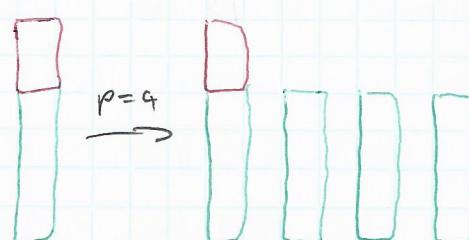
- More CPUs allow to solve bigger problems

↳ How much of work you can do in the same time with more CPUs

- f is the serial fraction of the work / p ist # CPUs

$$W_p = \underbrace{p \cdot (1-f)}_{\text{Parallel work time}} \cdot T + \underbrace{f \cdot T}_{\text{Serial work time}}$$

$$S_p = \frac{W_p}{W_1} = \underbrace{f}_{\text{Serial}} + \underbrace{p \cdot (1-f)}_{\text{Parallel}}$$



Work partition and scheduling

- Partition by programmer
- Scheduling by CPU → goal: full utilization
- Goldene Mitte zwischen small granularity ↔ Overhead
 - ↳ Scheduler muss mehr fragen, ob sie noch was zu tun haben
- Task parallel programming
 - Rekursiver code
 - Execute code → spawn other tasks → Wait for their result
 - Task Graph
 - Kanten: Abhängigkeiten
 - Knoten: Tasks
 - DAG (=Directed, acyclic graph)
 - Breite: Parallelismus
 - Tiefe: sequentieller Teil
 - Critical path: W_{ser} (=Längster Pfad zum Ziel)
 - T_p depends on scheduler
 - T_1 and T_{∞} are fixed
 - critical path
- Scheduler
 - Greedy scheduler theorem: $T_p \leq \frac{T_1}{P} + T_{\infty}$
 - Was ist, wenn Tasks verschiedene Laufzeiten haben?
 - "Work stealing" = Idle CPU steals work
 - "Work dealing" = Busy CPU spreads its work

parallel CPUs mit P work
Seriell kann nicht minimiert werden

Executor service

- Java threads are quite heavyweight \rightarrow mapped to OS-Threads
- Schedule tasks on given threads (=Executor service)
- ExecutorService exs = Executors.newFixedThreadPool (#CPU)
- exs.submit(task)
 - \hookrightarrow public class Task implements Runnable { }
 - \hookrightarrow public class Task implements Callable <T> { }
- exs.shutdown() public <T> call() { return <T> }] code
- Callable nur kompatibel mit ExecutorService und nicht mit Thread
- Gut für unabhängige Tasks, rekursive Aufrufe würden zu Deadlocks führen, da Tasks aufeinander warten würden

Fork Join Pool

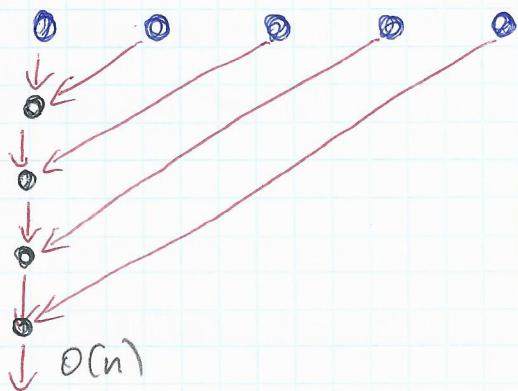
- Gut für rekursive Tasks, denn durch das Fork "kapselt" man einen rekursiven Task vom Elterntask ab, und joint sie dann wieder zusammen
- ForkJoinPool pool = new ForkJoinPool (#CPU)]
- rekursive Task temp = new rekursiveTask();] code
- Integer output = pool.invoke(temp)
- class rekursiveTask extends [RecursiveAction] recursiveTask<Integer>


```

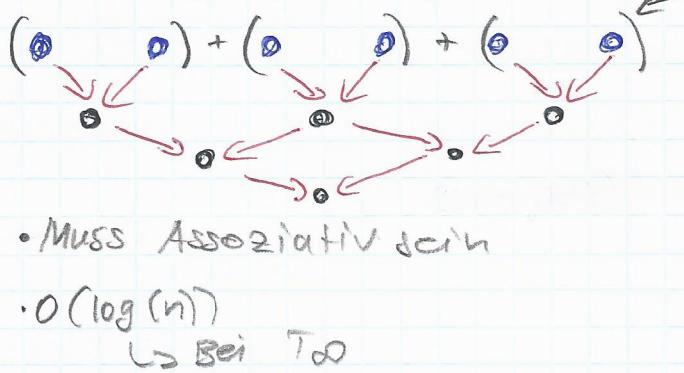
        Integer compute () {
          if (size <= cutOff) {
            //compute
          } else {
            new rekursiveTask(size/2)
              task1.fork() / task2.fork()
            return task1.join() + task2.join()
          }
        }
      
```

Rekursive Berechnungen

- Accumulator



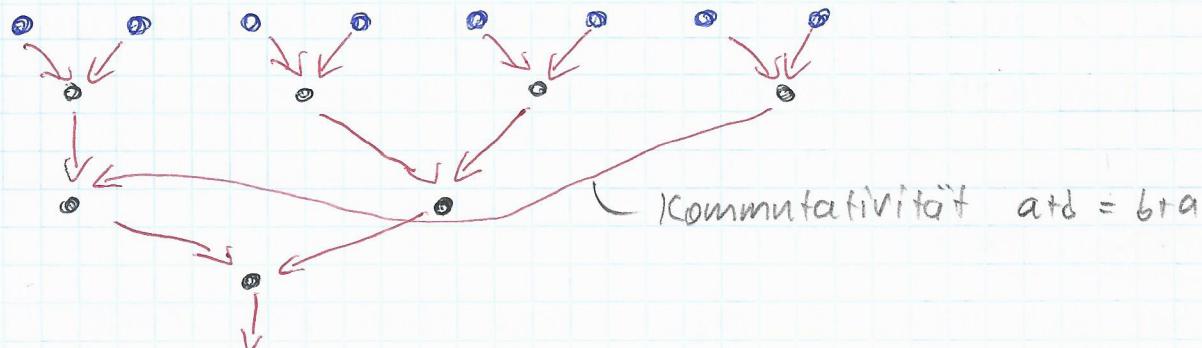
- Divide and Conquer



Assoziativ

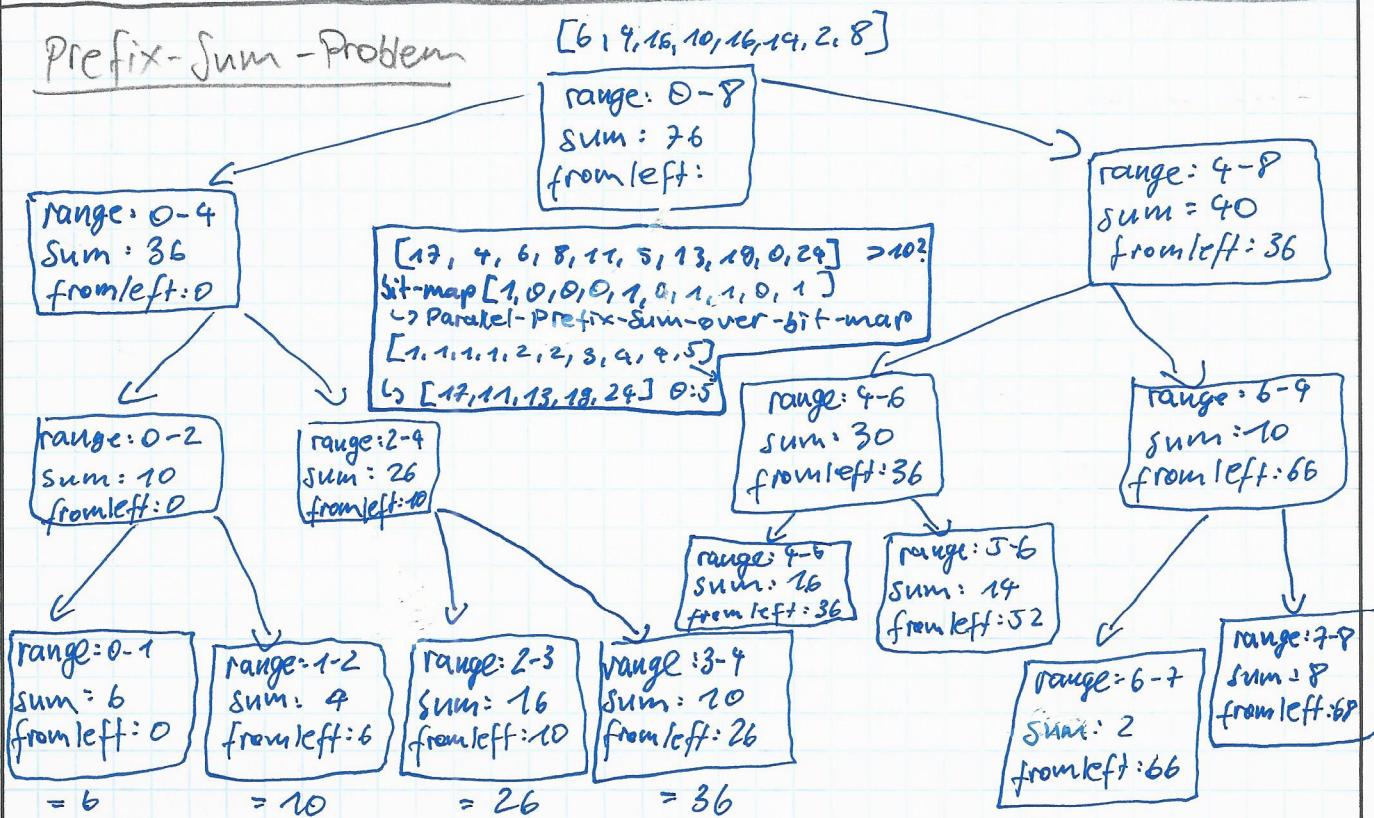
- Reduction

• Assoziativ + Kommutativ ($a+b = b+a$)



Prefix-Sum - Problem

Berechnet rekursiv: $O\left(\frac{n^2}{P} + T_{\text{BS}}\right) = O\left(\frac{n}{P} + \log(n)\right)$



Tiefe: $\log_{(2)}(8) = 3$

Locks

- Mechanism to ensure exclusive access to critical section
- Atomically checks, if lock is held (=1), if not (=0), set it to 1
- Safety-Property: only one thread at a time can be in the critical section
- Liveness: Acquiring the lock must terminate in a finite time, if the lock is not held

- Lock lck = new ReentrantLock();

lck.lock();

/code

lck.unlock();

Reentrant / recursive Lock

- Has a counter
 - If count=0, the lock is unlocked
 - Synchronized-Blocks are as well reentrant-locks
- Often use try { ... } finally { ... } to avoid forgetting to release the lock, if there's an exception

Race Conditions

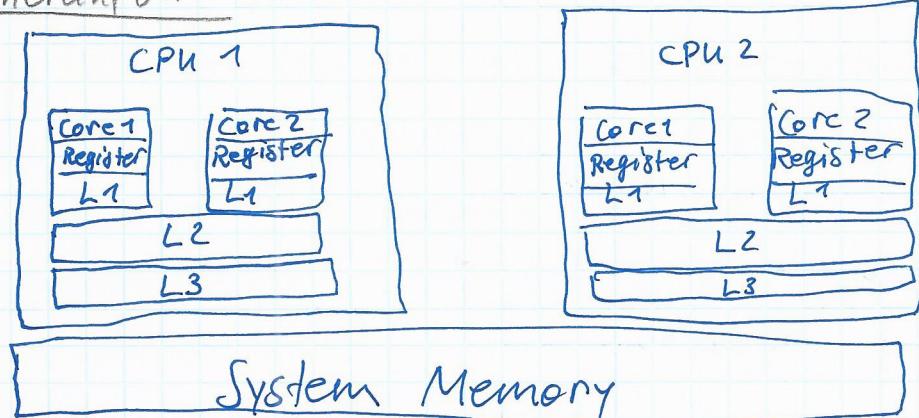
- Data race

- If you have a program, that runs parallel and access the same memory location at the same time with at least one thread writing
- Then, the accesses are non-deterministic
- The computation may give different results
 ↳ Wettkampf zwischen zwei Signalen, die als erstes Ausgabe beeinflussen wollen



- Wenn Java synchronized sieht, wird nichts mehr ungeordnet
 ↳ Relativ langsam

- Speicheranbau



Java Volatile

- (+) Prevents compiler to reorder the instructions
- (-) Doesn't deal with atomicity

- private volatile int x = 0;

- Jedes Mal, wenn ich Variable anfasse, wird sie geladen
- Schneller als synchronized, aber immer noch nicht gut, da jeder Thread seine Kopie abgleichen muss

Peterson's Lock

- Richtiger Mutex-Lock selbst implementiert

- while (Thread P) ↪ I want to go into critical section

flag[P] = true ↪ I am the victim, the other thread

victim = P ↪ I can go first

while (flag[Q] & victim == P)

//critical section

flag[P] = false ↪ I am not interested any more

- Only two threads

Filter-Lock

- Extension of Peterson's Lock

- Jeder Thread hat ein Level im Filter (level[+])

- Um critical-section zu betreten, muss thread alle levels durchlaufen

- Für jedes Level gibt es einen Peterson Lock

- 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))

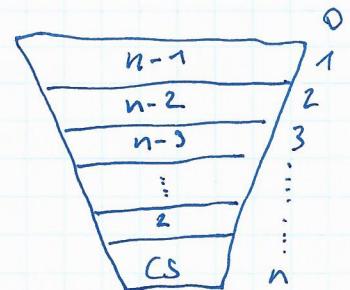
}

- Me as a thread can make progress

- Another thread wants to enter my level

- No more threads are in front of me

↳ Not fair!



for (int k=0; k < n; k++)

 if (k != me & & level.get(k) \geq lev)

 return true

Bakery Algorithmus

- Fair: First come, first serve
- Ich ziehe Ticket-Nummer, und komme der Nummer nach auch dran
- loop

number P = numberQ + 1

```
while(numberQ != 0 && numberQ < numberP); ] code
//critical section
```

numberP = 0

Atomic operations

- Test and Set

- Vergleicht nur 0/1 an gewählten Index
- Wenn 0, setze es zu 1, ansonsten return false

```
if(mem[input] == 0) {
    mem[input] = 1
    return true
} else
    return false ] code
```

- Compare and Set

- Wenn während dem ändern die Variable nicht verändert wurde, ersetze sie

```
oldval = mem;
if(old == oldval) ] code
    mem = new
return oldval
```

- GetAndSet

- Setzt Variable und gibt alten Wert zurück

Java atomic

- AtomicInteger test = new AtomicInteger(Integer.valueOf(10))
 test.set(Integer.valueOf(20))
 test.get() → return current
 test.compareAndSet(expect, update)
 test.getAndSet(newVal) → return old

SpinLock in Java

- while(state.getAndSet(true)) {}
- Ich spinne so lange, bis getAndSet-Methode false zurückgibt, und ich aus der Schleife springe
- Geht auch mit TestAndSet, aber nicht mit compareAndSwap.
- Um Spinnen zu verkürzen → Backoff
- Kein „first in first out“
- Kann passieren, dass Thread ungünstig fragt in Wait
 Nach Misserfolg
 gehe ich zufällige Zeit
 ↳ kommt nie dran

Semaphores

- Hat im Gegensatz zum Lock mehrere Zustände
- Nummer des Semaphores gibt an, wieviel slots frei sind

```

- acquire()
  synchronized(monitor)
    while(count <= 0)
      monitor.wait();
  count--;
  
```

```

  release()
  synchronized(monitor)
    count++;
    monitor.notifyAll();
  
```

code

- Semaphor mit Nummer=1 ist einfach ein Lock
- Rendez-Vous

- Special case of barrier with only two threads waiting for each other

P ↪ Lock semaphore
 P-arrived = 0 P-arrived

```

  release(P-arrived)
  acquire(Q-arrived)
  ...
```

Q ↪ Lock semaphore
 Q-arrived = 0 Q-arrived

```

  acquire(P-arrived)
  release(Q-arrived)
  ...
```

Barriers

- Generalisierung eines Rendez-Vous mit n-Threads
 - Kein Prozess kann die Barriere verlassen, bevor nicht alle anderen Prozesse die Barriere betreten haben
 - synchronized await() {
 count--
 while (count != 0)
 this.wait();
 this.notifyAll
}

Problems with Locks

- Deadlock: Each thread waits for each other
 - Starvation: Can't unlock lock
 - Livelock: Detect potential deadlock, but make no progress while trying to resolve deadlock

Monitors

- If a condition does not hold: Release monitor lock \rightarrow wait
 - Wait for condition to become true
 - Provides, in addition to simple locks, a mechanism to check conditions synchronized (monitor)
 - Immer in Verbindung mit while (condition)
`monitor.wait();`
 - Monitor ist Objekt mit wait, notify
 - Synchronized-Monitor hat $\overbrace{\text{immer in synchronized Block}}$
jedoch nur eine condition

Mehrere Conditions auf einer Lock

- Lock lck = new reentrantLock()
- Condition notify = lck.newCondition()
 - notify.await()
 - notify.signal()
 - notify.signalAll()
- Signal wird jedoch immer gesendet, auch wenn keine Threads warten

Sleeping Barber

- Additional counter for checking, if threads are waiting

Readers-Writer-Lock (Assignment 11 → RWlock)

- Mehrere Leser gleichzeitig \Leftrightarrow 1 Schreiber gleichzeitig
- Problem: Es gibt immer mehr Leser als Schreiber, deshalb kommt es als Schreiber nur selten dran
- X Readers \rightarrow 1 Writer \rightarrow X Readers \rightarrow 1 Writer
 - writers/readers (= # of writers/readers in CS)
 - writersWaiting/readersWaiting (= # of waiting threads)
 - writersWait (= Amount of readers to read until one writer can write)
 - acquire-read must wait, when
 - writers > 0
 - writersWait ≤ 0 \wedge writersWaiting > 0
 - acquire-write must wait, when
 - readers > 0 \wedge writers > 0
 - writersWait > 0

Coarse-Grained Locking

- Locks the whole list

⊕ Simple, Safe

⊖ Performance → serial

Fine-Grained Locking

- Jedes Element eigene Lock

- Hand-over-Hand in LinkedList

↳ ⊖ Man kann sich gegenseitig nicht überholen

Optimistic Locking

1. Finde Node ohne Locking ⊕ schnelles Traversieren

2. Locke Elternknoten und gefundenen Knoten

3. Validiere

4. Mache Änderungen

⊖ Man muss Liste zwei Mal durchlaufen: Finden + Validieren

Lazy Locking

- Wir implementieren „logisch-deleted“-Marker

1. Finde Knoten ohne Locking

2. checken, ob Nodes als deleted markiert sind

3. current als deleted markieren (schnell)

4. physikalisch umhängen (langsam)

⊕ contains-Methode, die am meisten verwendet

wird, muss nicht gelockt werden

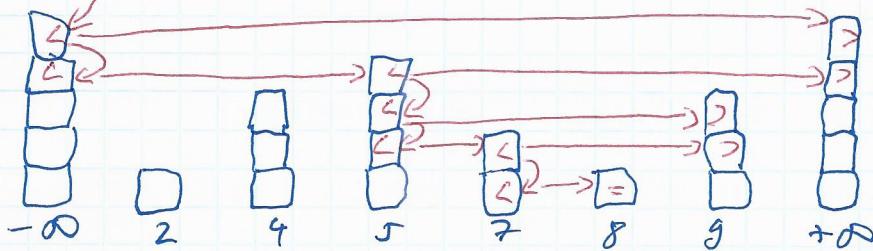
⊕ Nur ein Mal traversieren

lock

Lazy Skiplists

- Höhere Levels sind immer in niedrigen drin
- Je weiter man nach unten geht, desto mehr Listenelemente gibt es
- Level werden zufällig gewählt: $\Pr[\text{height} = n] : 0,5^n$
 ↳ Suche wird fast wie im Baum $\Rightarrow O(\log(n))$

(8)



Progress Conditions

	Non blocking	blocking
Everyone makes progress	<u>Wait-free</u> • Das Beste • Schwierig, zu realisieren	<u>starvation free</u>
Someone makes progress	<u>Lock-free</u> • Ingend ein Thread macht immer Fortschritt	<u>Deadlock-free</u>

Kein Sperren der critical Section

critical section gelockt

- Wait-free: Alle Operationen aller Threads werden ausgeführt, unabhängig von den Anderen
- Lock-free: Keine Operationen werden aufgehoben, aber durch Überschneidungen (cas) können Verzögerungen auftreten
- Speicherplatz \Leftrightarrow Performance
 - Je mehr Speicherplatz, desto besser die Performance

ABA-Problem in CAS

- A liest alten Status
- Anderer Thread ändert Status
- Thread B liest geänderten Status
- Anderer Thread ändert Status zum Alten
- Thread A überprüft und schreibt neuen Status

→ Kann durch Pointer Tagging verhindert werden, bei jeder Benutzung inkrementiere ich my-wed-counter
 → Garbage Collector } Passiert wegen gleichen Pointers, nicht Daten
 } Bits in Reference
 } 32 Versionen

Lock-Free (=optimistisch)

- Wird von „Compare and Swap“ implementiert
- Lock-free ist nicht automatisch besser als sein blocking equivalent, da CAS auch „Spins“
 ↳ Mit Backoff wird es besser
- Bei vielen Kollisionen → Lock

1. Ich kopiere aktuellen, globalen Zustand zu lokaler Variable
2. Ich mache auf lokaler Variable die Änderungen
3. Mittels CAS schaue ich, ob globale Variable nicht verändert
 - 3.1. Wenn ja, versuche ich das Ganze nochmals
 - 3.2. Wenn nein, lokal → global

↳ do { ... } while (!x.compareAndSet (old, new))

- Double Compare And Set (=DCAS)

- Atomic Markable Reference

um Knoten physikalisch zu löschen

- boolean compareAndSet (V expected Reference, V new Reference,

boolean expectedMark, boolean newMark)

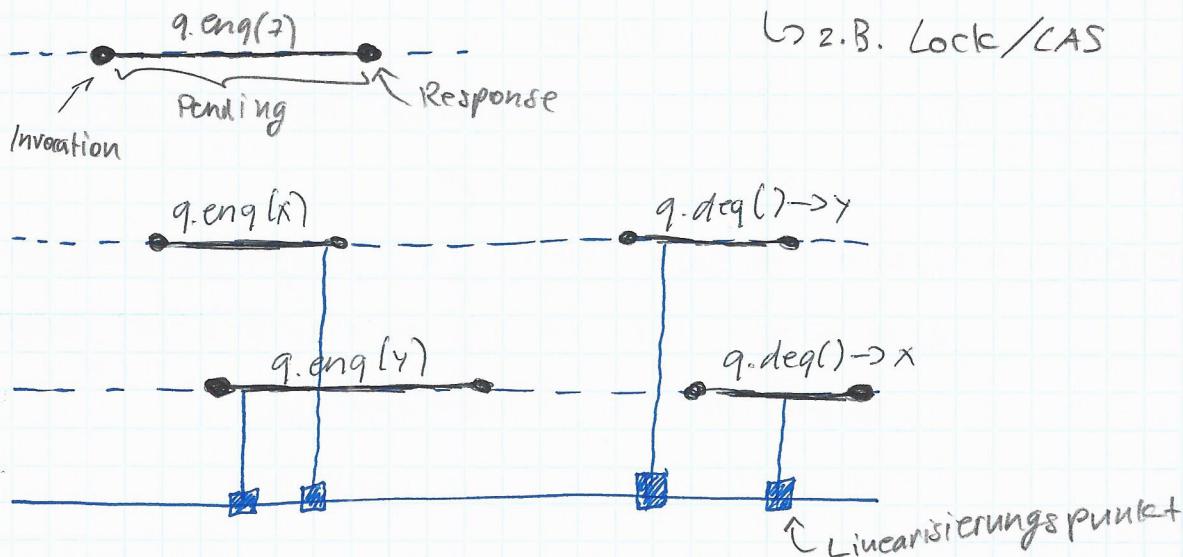
um Knoten logisch zu löschen

- Knoten helfen sich gegenseitig, wenn Thread stirbt, und

- Knoten nur logisch gelöscht ist, ich traverse, finde, lösche physikalisch

Linearisierbarkeit

- Methode ist linearisierbar, wenn ihr Resultat auffassbar sichtbar für alle anderen Threads wird



- History: Sequence of Invocation + Response

A $q.enq(3) \hookrightarrow$
A $q.void \leftarrow$
A $q.enq(5) \leftarrow$ pending: no response
A $q.enq(5) \leftarrow$ pending: no response
B $p.enq(4) \hookrightarrow$
B $p:void \leftarrow$
B $q.deq \hookrightarrow$
B $q:3 \leftarrow$

↳ darf mit $complete(H)$ komplettiert werden

} Subhistory is complete, no pending responses

Projections

$\cdot H/p : B p.enq(4)$ } Objekt p
 $B p:void$

$\cdot H/B : B p.enq(4)$ } Thread B
 $B p:void$
 $B q.deq()$
 $B q:3$

- Sequential History: Wenn Methoden von Threads nicht überlappen
- Well formed History: Wenn per Thread Projektion sequentiell ist
- Equivalent History: Ausführreihe/Reihenfolge anders, aber Ausgabe bleibt gleich
- Precedence: Wenn zuerst Thread A response bekommt, und dann Thread B eine invocation macht
 - ↳ Gegenteil von Overlapping
 - ↳ A wird vor B ausgeführt
$$m_0 \xrightarrow{H} m_1 \quad (m_0 \text{ precedes } m_1)$$

(= m_0 geht m_1 voran)
- Linearizability: Wenn history A erweitert wird zu history B, so dass B äquivalent ist zu einer sequentiellen history S
 - $\xrightarrow{S} C \xrightarrow{S}$
 - Wir erweitern so, dass jedes Objekt projiziert/isoliert angeschaut, ebenso linearisierbar ist
- Modularität: Wenn zwei Objekte unabhängig voneinander linearisierbar sind, sind sie es zusammen auch

Sequentielle Konsistenz

- Schwächer als Linearisierbarkeit
- Man darf Reihenfolge im Thread nicht ändern, aber Reihenfolge zwischen Threads
- Nicht modular

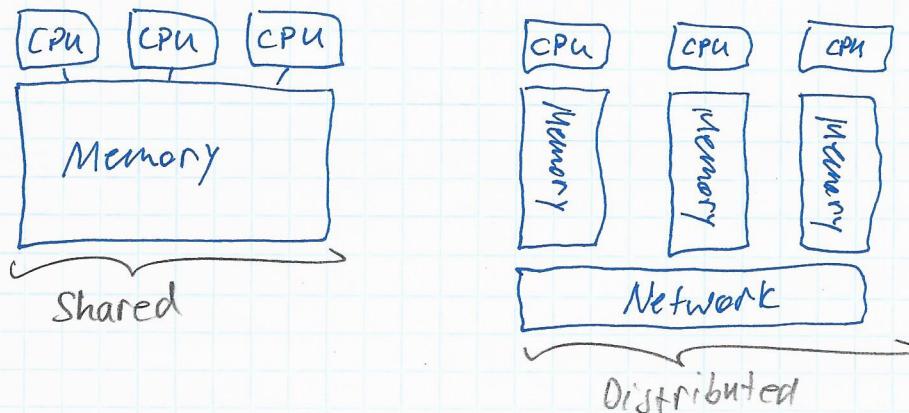
Consensus Protocol / Number (= Übereinstimmung)

- Agreement among a number of processes for a single data value
- The processes must communicate with each other, and agree on a single value
- Consensus Number: Wie viele Prozesse
- Atomic registers: Consensus Number 1
- Compare and Set: Consensus Number 0

Transactional Memory

- Programmierer definiert atomic-code-section
- Optimistische Lock (Ich lasse alle rein und prüfe später)
- Transaktion macht Snapshot vom gesamten Speicher und arbeitet dann lokal darauf
- Ich überwache alle meine Änderungen/Lesoperationen auf lokaler Speicher und committe, ob diese Dinge global verändert wurden

Distributed Memory & message passing



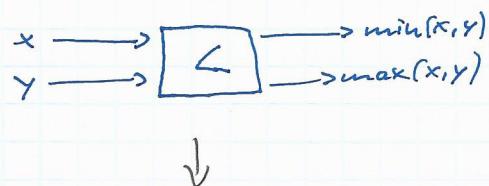
- Synchronous: Sender waits, until message is received from other system
- Asynchronous: Sender does not wait, places message into buffer for receiver to get it

Message Passing Interface (=MPI)

- Jeder Prozess hat eigene Rank-Nummer
- Size = Anzahl Prozesse im System
- Prozess mit Rank=0 ist der Chef und gibt Lösung dann zurück

Sorting Networks

- Basiselement: Comparitor



- Statische Struktur

- Wenn Netzwerk nur 0/1



sortieren kann, dann kann y
es auch andere Zahlen sortieren

↳ Korrektheitsbeweis

- Damit kann man auch insertion-sort darstellen

Presburger sets & Relations

- Benutzt für automatische Loop-Vectorization

- Loop muss unabhängig sein

- Presburger sets: Ich will Schleife als Punkte im Vektorraum darstellen \rightarrow Illustration von Speicherzugriffen

- Presburger Relationen: Illustration von Abhängigkeiten

↳ Semantik eines Programms wird statisch modelliert

Scheduler (must be good)

- Loop reversal: Loop wird umgedreht

- Loop interchange: innerer Loop \leftrightarrow äußerer Loop

- Loop fusion: zwei Loops werden vereinigt

- Loop distribution: Ein Loop wird zu zwei Loops