## Moore’s Law, Web Scale, and the Mess We’re in:

S.2:

* Moores Law hat nicht mehr standgehalten -> Erhöhung der Rechenkapazität durch mehrere Kerne
* Amdahl’s Law beschreibt, wie hoch der theoretische Performancegewinn bei Parallelisierung von Workloads ist (ob es sich lohnt ein Programm zu parallelisieren

S.3:

* Probleme die leicht parallelisierbar sind, müssen so strukturiert werden, dass sie auf mehreren CPUs, oder sogar mehreren Maschinen laufen, um einen Performancegewinn zu erzielen -> sie müssen also horizontal skalieren

## Why is Concurrency Hard?

S.4:

- Race Conditions: Tritt auf, wenn zwei oder mehr Operationen in einer bestimmten Reihenfolge erfolgen müssen

- wenn das Programm jedoch so geschrieben ist, dass diese Reihenfolge garantiert ist, kann es zu unterschiedlichen Ergebnissen kommen

- meistens sieht man das anhand eines sog. Data Race, wobei eine nebenläufige Operation eine Variable ausliest während eine andere die Variable beschreibt

- das Beispiel von S.4 mitnehmen

S.5:

- Beim Beispiel wird entweder nichts, 0 oder 1 ausgegeben je nachdem in welcher Reihenfolge das Programm abläuft

- erwähnen, dass man all diese Szenarien immer wieder im Kopf durchgehen muss beim Programmieren von nebenläufigen Programmen

S.6:

- Manche Data Races bleiben lange unentdeckt, weil es erst ab einem gewissen Skalierungslevel auftritt (Beispiel bringen)

- hier zusätzlich noch erwähnen, dass Data Races oft auftreten, weil Entwickler den Code den sie schreiben als sequentiell betrachten und ein Aufruf einer Goroutine die eine Variable ändert , ganz viele Aufrufe dazwischen, und dann die Prüfung auf die von der Goroutine geänderte Variable kann oft in der richtigen Reihenfolge ausgeführt werden, aber es gibt eine kleine Chance, dass ein Data Race auftritt

## Atomicity

S. 6-7:

- Ob etwas atomar ist hängt von dem Kontext ab (Operationen auf einer Variablen, die nur innerhalb eines Threads verwendet wird, ist atomar, bei mehreren Threads jedoch nicht)

- Operationen atomar zu machen schützt vor den Nebenwirkungen der Nebenläufigkeit

- Operationen, die atomar scheinen, sind oft gar nicht atomar (i++)

## Memory Access Synchronization:

S.8:

* Teil eines Programmes welches exklusiven Zugriff auf eine geteilte Ressource benötigt nennt man „Critical Section“
* Im Beispiel 3 kritische Sektionen:
  + Goroutine welche data inkrementiert
  + If Statement check auf data
  + Print Statement welches data für den Output holt

S.8-9:

- Lock Beispiel übernehmen

S.9:

- immer wenn man exklusiven Zugriff auf eine Variable haben möchte muss diese Locked und anschließend Unlocked werden

- Data Race besteht immer noch: Nur bei der If-Condition zustimmen und anschließend trotzdem 1 ausgeben ist verhindert durch Locks, Nichtdeterminismus zwischen Ausführung des If und des Else Blocks bleibt

- Lösung des Speicherzugriffs durch Locks bildet neue Probleme

## Deadlocks, Livelocks and Starvation

Deadlock:

S.10:

* Alle nebenläufigen Programme warten aufeinander und keine Arbeit wird dabei verrichtet
* Kann ohne Eingriff von außen nicht recovern
* Eigenes Deadlock Beispiel bringen

S.12:

- Coffman Conditions als Definition für einen Deadlock

- Mutual Exclusion: Ein nebenläufiger Prozess hat exklusive Kontrolle über die Ressourcen die er benötigt

- Wait For Condition: Ein nebenläufiger Prozess hält bereits Ressourcen während er auf Ressourcen wartet, die von einem anderen nebenläufigen Prozess fertig verwendet wurden

- No Preemption Condition: Die Freigabe von Ressourcen kann nicht erzwungen werden bis die Ressourcen fertig verwendet wurden

- Circular Wait Condition: Es gibt eine zyklische Kette aus nebenläufigen Prozessen, die Ressourcen halten, die vom jeweils nächsten Prozess benötigt werden

Livelocks

S.13:

- nebenläufige Operationen, die aber den Zustand des Programms nicht weiter fortschreiten lassen

S.15:

* Grund warum Livelocks auftauchen: Zwei oder mehr nebenläufige Prozesse versuchen ohne Koordnination einen Deadlock zu verhindern
* Livelocks schwerer zu finden als Deadlocks, da es weiterhin so aussieht als würde das Programm Arbeit verrichten
* Livelocks Subset von Starvations

Starvation

S. 16:

- Starvation ist jede Situation wo ein nebenläufiger Prozess nicht genügend Ressourcen bekommt um zu funktionieren

- Beim Livelock war die fehlende Ressource ein Shared Lock für die Direction

- Bei der Starvation geht es darum, dass es Prozesse gibt, die unfairerweise andere Prozesse davon abhalten Arbeit (so effizient wie möglich) zu verrichten

S.17:

- An dem Beispiel kann man ebenfalls sehen, wie synchronisierte Speicherzugriff zu Einbußen in der Performance des Prozesses führen kann (würde man den Lock im Greedy Prozess öfter freigeben, müsste man auch öfter auf den Lock warten, wodurch der Prozess langsamer wird, mit weniger Locks wird der Greedy Worker zwar schneller, verhindert aber andere Prozesse)

S. 18: Starvation kann einen Prozess auch komplett zum Stillstand bringen – z.B. wenn ein Greedy Worker eine Ressource lockt, die ein anderer Prozess auch braucht, sie aber nie wieder freigibt -> dann wartet der andere Prozess einfach ewig

## The Difference Between Concurrency and Parallelism:

S.23

* Nebenläufig und Parallel werden oft synonym verwendet für „etwas was zur selben Zeit läuft wie etwas anderes“ , obwohl es zwei unterschiedliche Dinge sind
* Es hängt vom Fall ab, ob etwas sowohl nebenläufig als auch parallel ist oder nur eines der zwei Dinge
* Concurrency is a property of the code; parallelism is a property of the running program

S.24:

- Um die Nebenläufigkeit kümmert sich der Entwickler; ob die Prozesse wirklich parallel laufen, hängt von dem Kontext ab (hat meine Maschine nur einen oder mehrere Kerne?)

Struktur:

* Was ist Nebenläufigkeit?
* Was unterscheidet Nebenläufigkeit von Parallelismus?
* Warum ist Nebenläufigkeit wichtig?
* Warum ist Nebenläufigkeit so schwer?
  + Race Conditions
    - Data Races
    - Atomicity
    - Memory Access Synchronization
  + Erklärung des Problems der speisenden Philosophen
  + DeadLocks anhand des Problems