## 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

Text:

Man bezeichnet ein System als nebenläufig, wenn es dazu in der Lage ist mehrere Tasks simultan, sei es in Form von unterschiedlichen Prozessen oder Threads, abzuarbeiten. Bei dieser Definition wird oft fälschlicherweise angenommen, dass die Nebenläufigkeit der Parallelität gleichzusetzen ist. Allerdings heißt Nebenläufigkeit jedoch nicht gleichzeitig, dass die Tasks auch parallel bearbeitet werden. Ein System kann nämlich auch dann schon als nebenläufig bezeichnet werden, wenn es lediglich mehrere Tasks in einer Warteschlange hält und zwischen diesen hin und her wechselt, bevor diese vollständig bearbeitet wurden.

Genau so macht es ein Betriebssystem, welches auf einer Maschine mit nur einem Prozessorkern läuft. Es definiert Zeitschlitze, in denen jeweils ein Prozess Zugriff auf den Prozessorkern hat, bevor der nächste Prozess an der Reihe ist. Ob der jeweilige Prozess fertig bearbeitet wurde, ist keine Bedingung dafür, dass der nächste Prozess bearbeitet werden kann. Unfertig bearbeitete Prozesse werden wieder in die Warteschlange eingereiht.

Allerdings sind Systeme, die Tasks parallel bearbeiten, auch gleichzeitig nebenläufig. Angenommen die Maschine, auf der unser Betriebssystem läuft, hat nun mehrere Prozessorkerne zur Verfügung. Damit hat unser Betriebssystem die Möglichkeit so viele Tasks gleichzeitig bearbeiten zu lassen, wie es Prozessorkerne gibt. Mit der gewachsenen Anzahl der Prozessorkerne und die dadurch ermöglichte Parallelität verfügt das System auch über die Fähigkeit die gleiche Anzahl an Tasks in viel kürzerer Zeit zu bearbeiten. Die parallele Bearbeitung bedeutet also einen Performancegewinn.

Dies ist jedoch nicht der einzige Vorteil den Nebenläufigkeit mit sich bringt. Man stelle sich zum Beispiel einen Browser vor über den der User durch Eingabe einer URL zu einer Website gelangt. Nach Eingabe der URL und betätigen der Enter-Taste bemerkt der User, dass das Laden der Website sehr lange dauert und entscheidet sich stattdessen eine andere Website zu besuchen. Die Eingabe der anderen URL ist dem User nur möglich, weil das Laden der Website und das Erkennen von User-Eingaben im UI parallel, oder zumindest nebenläufig, bearbeitet wird. Ansonsten würde der User warten müssen, bis die Website vollständig geladen ist oder ein Fehlerfall auftritt. Neben dem Performancegewinn bietet die Nebenläufigkeit also auch noch den Vorteil der Reaktionsfähigkeit.

Nebenläufigkeit kann auch dabei helfen eine Anwendung skalierbarer zu machen. Ein Beispiel dafür sind Webserver, die die Anfragen von mehreren hundert Clients gleichzeitig bearbeiten müssen. Durch die Nebenläufigkeit können auf unterschiedlichen Prozessorkernen mehrere Anfragen parallel verarbeitet werden. Falls eine dieser Anfragen beispielsweise einen länger dauernden I/O-Zugriff ausführt kann während der dadurch auftretenden Wartezeit die Anfrage eines anderen Clients bearbeitet werden. Nebenläufigkeit stellt also auch sicher, dass die zur Verfügung stehenden Ressourcen möglichst effizient ausgeschöpft werden.

Doch der Einsatz von Nebenläufigkeit bringt auch Gefahren mit sich. Ein Beispiel dafür ist der folgende in der Programmiersprache Go geschriebene Code. Die Funktion „race\\_condition“ verfügt über eine Variable „data“, die auf den Wert 0 geprüft wird. Wenn der Wert 0 ist, soll der Wert zurückgegeben werden. Wenn der Wert jedoch nicht 0 ist, wird -1 zurückgegeben. In den Zeilen 3 – 5 startet die Funktion einen neuen Thread, der die Variable „data“ um eins erhöht. Bei Ausführung der Funktion „race\\_condition“ ergeben sich drei mögliche Ausgabewerte. Die beiden offensichtlichsten Ausgabewerte sind 0 und -1. Wenn der nebenläufige Thread aus den Zeilen 3-5 es noch nicht geschafft „data“ um eins zu erhöhen wird 0 ausgegeben. Der Wert -1 wird zurückgegeben, wenn der nebenläufige Thread es vor der Prüfung auf 0 geschafft hat „data“ hochzuzählen. Der dritte Fall ist weniger offensichtlich und gleichzeitig sehr problematisch. Er tritt ein, wenn die Prüfung von „data“ auf den Wert 0 erfolgreich ist, jedoch zwischen der Prüfung und der Rückgabe von „data“ der nebenläufige Thread das Inkrement um eins durchgeführt hat. In diesem Fall gibt „race\\_condition“ 1 zurück.

Um zu beweisen, dass bei jeder Ausführung völlig unklar ist welches Ergebnis zurückgegeben wird, wird in der Main-Funktion die Race Condition 100.000 mal ausgeführt. Die Funktion „print\_shares“ in Zeile 18 nimmt die Liste der Ausgaben von „race\\_condition“ entgegen und gibt die Anteile der Ergebnisse -1, 0 und 1 in der Konsole aus. Für eine Ausführung dieses Programms ergibt sich beispielhaft die folgende Ausgabe:

Hier ist zu erkennen, dass vor allem der Fall, bei dem „race\\_condition“ 0 zurückgibt, auftritt. Aber auch die anderen beiden Fälle treten auf. Besonders ärgerlich ist, dass der Fall, bei dem 1 zurückgegeben wird, auftritt und gleichzeitig sehr selten ist. Damit das Programm in die Zeilen 7 und 8 gelangen darf, muss die Variable „data“ den Wert 0 enthalten. Allerdings hat der nebenläufige Thread durch die simulierte Arbeit in Zeile 7 manchmal genügend Zeit den Inhalt von „data“ vor der Rückgabe zu ändern

Ein solcher Concurrency Bug nennt sich Race Condition. Genauer gesagt handelt es sich hier um einen sog. „Data Race“, da es darauf ankommt, welcher Thread seine Arbeit mit den Daten in der Variable „data“ verrichtet. In Unit-Tests wird ein solcher Concurrency Bug, nicht erkannt, da er, wie in der Konsolenausgabe zu erkennen ist, dafür viel zu selten auftritt. In Produktionsumgebungen jedoch, wo eine solche Funktion tausende Male am Tag ausgeführt wird, kann ein solcher Bug auftreten. Dieser ist dann aber schwer oder gar nicht zu replizieren, da er so selten vorkommt. Bei Concurrency Bugs kann es auch vorkommen, dass sie erst ab einem gewissen Skalierungsniveau auftreten. Beispielsweise wird ein Dienst, der einen solchen Bug enthält, 1-mal pro Tag aufgerufen. Wenn die Wahrscheinlichkeit, zu der der Concurrency Bug auftritt, die gleiche ist wie in unserem obigen Beispiel kann dieser Bug jahrelang unentdeckt bleiben. Wenn die Anfragen auf den Dienst jedoch zunehmen, kommt es umso häufiger vor, dass dieser Bug auftritt und wohlmöglich Inkonsistenzen in den Daten und Ausgaben des Dienstes verursacht.

Um Race Conditions zu verhindern, können Zugriffe auf Variablen, die sich während der Ausführung eines bestimmten Abschnitts im Code nicht verändern dürfen, synchronisiert werden. Dies geschieht durch die Verwendung eines sog. „Mutex“. Nebenläufige Threads können sich durch das Erwerben eines Locks auf einen Mutex den Zugriff auf eine Variable, die zu dem Mutex gehört, reservieren. Angewandt auf das obige Beispiel sieht der Code der Funktion „raceCondition“ folgendermaßen aus:

Hier ist in den Zeilen 5 - 7 zu sehen, dass der nebenläufige Thread den Mutex zuerst sperrt, bevor er „data“ inkrementiert. Anschließend wird der Mutex wieder freigegeben, da die Änderung an der geteilten Variable vollständig durchgeführt wurde. Der Main-Thread hingegen sperrt in Zeile 9 den Mutex, um sicher zu gehen, dass sich der Inhalt von „data“ in den Zeilen 10 – 12 nicht verändert.

Wenn nun die gleiche Main-Funktion wie bereits in dem vorigen Code-Beispiel ausführt, werden folgende Ergebnisse in die Konsolenausgabe geschrieben:

Wie anhand der Konsolenausgabe zu erkennen ist, gibt es keinen einzigen Fall mehr, bei dem sich der Wert von „data“ nach der Prüfung auf 0 doch noch zu 1 ändert. Das Reservieren und Freigeben eines Mutex sichert also die Konsistenz einer Variable innerhalb einer kritischen Sektion. Der Begriff „kritische Sektion“ bezeichnet dabei einen Abschnitt im Code, der Lese- oder Schreiboperationen auf eine zwischen Threads geteilte Variable ausführt.

Allerdings bringt diese Methode der Konsistenzsicherung weitere Probleme mit sich. Eines dieser Probleme ist der sog. Deadlock. Er tritt dann auf, wenn alle Threads eines Programms zur Bearbeitung ihres Tasks einen Mutex reservieren müssen, der aber bereits von einem anderen Thread gehalten wird. Das bedeutet, dass im Falle eines Deadlocks alle Threads aufeinander warten und keine Arbeit verrichtet wird. Ein einfaches Beispiel eines Deadlocks zeigt der folgende Code:

Im obigen Codebeispiel gibt es zwei Threads. Beide Threads benötigen die Variablen a und b innerhalb einer If-Bedingung. Um zu sichern, dass die Werte, auf die die Variablen geprüft wurden, innerhalb der If-Bedingung gleichbleiben, gibt es für jede Variable einen Mutex. Nun ist es so, dass die beiden Threads eine unterschiedliche Reihenfolge haben den Mutex zu reservieren. Dies kann dazu führen, dass der nebenläufige Thread erst den Mutex für a und der Main-Thread den Mutex für b reserviert. Im nächsten Schritt versuchen beide Threads den Mutex, der vom jeweils anderen Thread bereits reserviert wurde, selbst zu reservieren. Da dies jedoch nicht möglich ist kommt es zu einem Deadlock, da alle Threads warten. Infolgedessen stürzt das Programm ab.

Natürlich können nicht nur mit zwei Threads Deadlocks erzeugt werden. An einem Deadlock können beliebig viele Threads beteiligt sein. Das Beispiel aus Listing 5 lässt sich auf drei Threads erweitern, indem eine weitere Variable und ein weiterer Mutex hinzugefügt wird. Ein solches Beispiel ist in Listing 6 zu sehen. Hier muss jeder Thread zwei Locks auf einen Mutex reservieren, bevor dieser seine Arbeit verrichten kann. Die Besonderheit hierbei ist, dass der zweite Lock in jedem Thread in einem der anderen Threads der erste Lock ist. Beispielsweise wird in Thread 1 in Zeile 11 als zweiter Mutex „lockB“ reserviert, wohingegen in Thread 2 in Zeile 21 „lockB“ als erster Mutex reserviert wird. Angenommen das Scheduling der drei Threads führt dazu, dass jeweils die erste Lock-Operation ausgeführt wird. Dann sind anschließend alle Mutexes reserviert und keiner der Threads kann im nächsten Schritt die benötigte Ressource reservieren. Schlussendlich kommt es wieder zu einem Deadlock und das Programm stürzt ab.