1.1 Programmiertechniken für geteilte Datenstrukturen mit fine-grained Concurrency

1) Methoden für Zugriff / Manipulation der Datenstruktur sind **memoryless** / es gibt keine persistente Information.

Warum?
↳ Selber brechnen die Methoden nichts, brauchen dafür also keine Variablen.
↳ Information über den Zustand der Datenstruktur ist ohnehin im nächsten Moment ungültig.

2) Der erste Schritt bei Zugriffsmethoden ist, Information über den Zustand der Datenstruktur aufzunehmen.
Dazu werden shared Pointer ausgelesen und ggf. von dort aus gesucht.
Bei push/pop genügt es, zu wissen, wohin g_tos zeigt, und ob g_tos NULL ist.

3) Dann wird lokal das Update der Datenstruktur vorbereitet.
Lokal heißt dabei, der Speicherbereich kann nicht von anderen Threads zugegriffen werden.
Bei push ist darin so eine lokale Vorbereitung.

4) Schließlich wird atomar die lokale Änderung gepublished
Beachte: Das lokale Update muss so gewählt sein, dass die gesamte Datenstruktur wieder in einen Zustand übergeht, den die Threads handhaben können.

5) Es wird optimistic Concurrency genutzt.
wir gehen davon aus, dass der in (2) ermittelte Zustand
bei der Veröffentlichung in (4) noch besteht.
Ist das nicht der Fall, beginnen wir erneut bei (2).
Das atomare Update muss also so beschaffen sein,
dass es die Gültigkeitsprüfung durchführen kann.
Mehr als compare_exchange (oder compare_and_swap, cas)
sollte man allerdings nicht nutzen wollen.

1.2 Das ABA-Problem

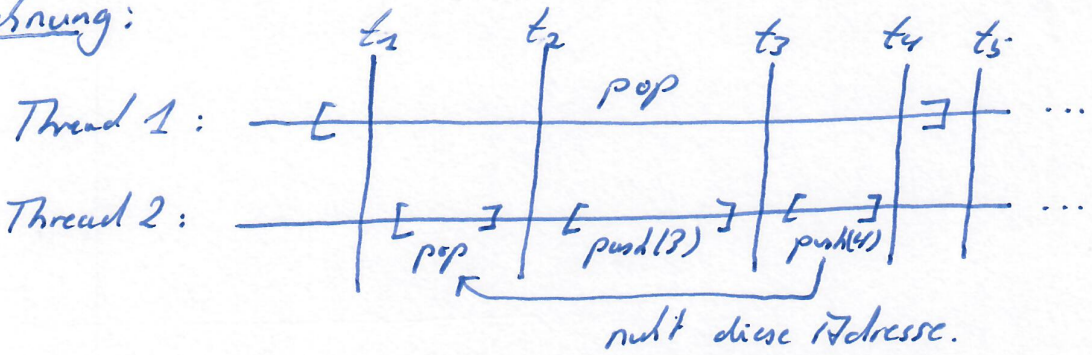
Problem: . Oben erwähnte Gültigkeitsprüfung.
. C++ vergisst freigewordenen Speicher erneut.

Genauer: . Ein Thread zeigt auf eine Zelle,
die Gestalt A hat.
Zum Beispiel gilt curtos == g_tos.
. Ein anderer Thread löscht diese Zelle.
. Ein dritter Thread alloziert die Zelle.
Sie hat nun Gestalt B, ohne dass Thread 1 das merkt.
. Im Laufe der Berechnung wird Gestalt A
wieder hergestellt.
. Nun führt Thread 1 die Gültigkeitsprüfung durch
und denkt, es sei nichts passiert.

Beispiel: Betrachte folgenden Stack-Zustand

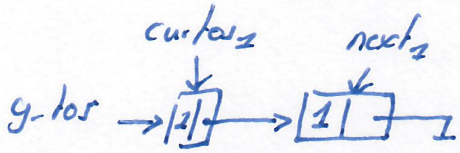
- Thread 1 möchte ein pop ausführen, ist aber langsam.
- Thread 2 macht pop. push(3). push(4), wobei push(4) die Zelle erhält, die beim pop von Thread 2 frei wird.

Berechnung:

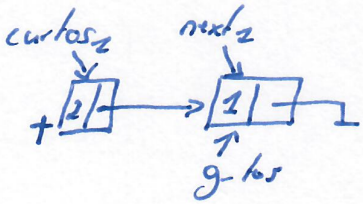


Zustände:

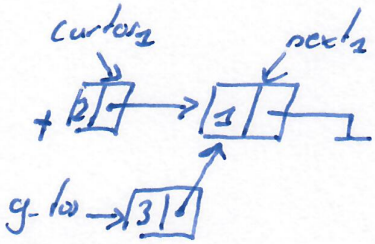
t_1:



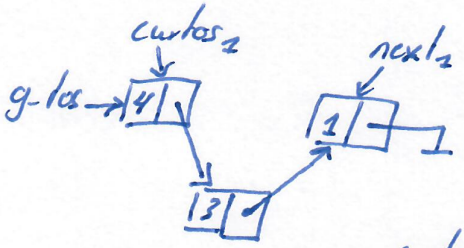
t_2



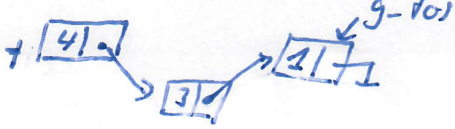
t_3



t_4



t_5



3-

Probleme: • Wir verlieren Stack-Elemente.
• Wir haben ein Memory-Leak.

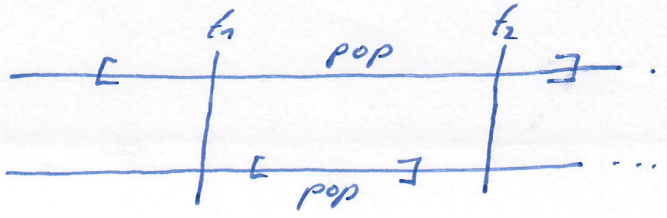
Lösung: • Statte jeden Pointer mit
einen Tag / Stamp / Modification-Counter aus,
der die Anzahl an ausgeführten Operationen zählt.
• Architekturen bieten dazu compare-exchange
für Pointer doppelte Breite / Bitzahl
(double width cas)
• Löst das Problem in der Theorie eigentlich nicht,
da es Überläufe gibt.
In der Praxis schon.

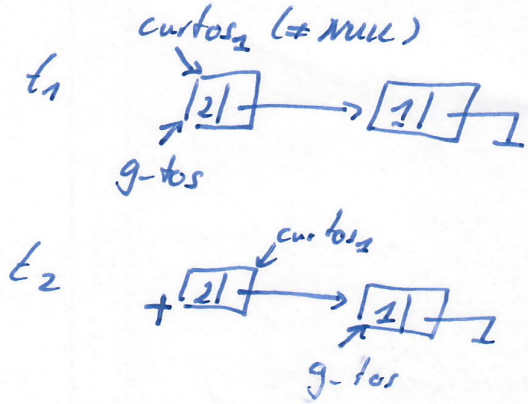
1.3 Segmentation Fault

Problem: . Trotz der Verwendung von Modification-Counter
kann es zu Segfaults kommen.

Beispiel: . Thread 1 führt ein pop aus und prüft curtos ≠ NULL.
Dann wird er unterbrochen.
. Thread 2 führt derweilen ein pop aus, bis zum Ende.
Dabei wird der Zelle gelöscht,
auf der curtos von Thread 1 zeigt.
• Nun greift Thread 1 auf curtos->next zu.

Berechnung:





Lösung:
- Einen einfachen Trick zur Lösung des Problems gibt es nicht.
- Standardlösungen implementieren nebenläufige Datenstrukturen ihren eigenen Safe-Memory-Reclamation (SMR) Algorithmus.
- Garbage-Collection ist ein möglicher SMR-Algorithmus. Tatsächlich hat unsere Implementierung von Treibers Stack unter Java keine Probleme.
- Es geht aber deutlich schneller als Garbage-Collection.

Idee: Informiere den SMR-Algorithmus über beabsichtigten Zugriff auf eine Zelle.
Schütze die Zelle vor Deletion.
↳ Hazard-Pointer

Problem:
- In welchem Sinn ist dann die parallele Implementierung korrekt?
- Korrektheit soll sagen:
  "Sieht irgendwie aus wie ein sequentieller Stack."
- Was soll das heißen?
  ↳ Linearisierbarkeit.