Informatik II: Algorithmen und Datenstrukturen SS 2013

Vorlesung 7b, Mittwoch, 5. Juni 2013

(Lokalität des Zugriffes, Cache- bzw. IO-Effizienz)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Lokalität / Cache- bzw. IO-Effizienz
  - Fortsetzung Laufzeitvergleich Feld vs. verkettete Liste
  - Lokalität
  - Wie funktioniert ein Cache (für CPU und Disk)
  - Neues Effizienzmaß: Anzahl **Block**operationen
  - Übungsblatt 7, Aufgabe 2: Anzahl Blockoperationen von MergeSort analysieren

Laufzeit Listen vs. Felder 1/4

UNI FREIBURG

- Laufzeit doppelt verkettete Liste
  - Einfügen an beliebiger Stelle: $O(1)$
  - Entfernen an beliebiger Stelle: $O(1)$
  - Zugriff auf i-tes Element der Liste: $O(\min\{i, n - i\})$
- Laufzeit dynamisches Feld
  - Einfügen am Ende: amortisiert $O(1)$ … siehe letzte VL
  - Entfernen am Ende: amortisiert $O(1)$ … siehe letzte VL
  - Zugriff auf i-tes Element der Liste: $O(1)$
  - Einfügen an i-ter Stelle: $O(n - i)$
  - Entfernen an i-ter Stelle: $O(n - i)$

Laufzeit Listen vs. Felder 2/4

UNI FREIBURG

- Zeitmessung in der Praxis
  - Beim Einfügen / Entfernen am Ende scheinen Liste und (dynamisches) Feld gleich gut zu sein

    Die Liste sieht sogar besser aus, weil **immer** O(1) und das dynamische Feld nur **im Durchschnitt** O(1)
  - Die Laufzeit wollen wir jetzt mal konkret nachmessen
  - Beobachtung:

    LinkedList ca. 5 mal langsamer als DynamicArray (für 10M Elemente) in JAVA

    in C++ : ca. 3 mal langsamer

Laufzeit Listen vs. Felder 3/4

UNI FREIBURG

- Analyse gemessener Laufzeitunterschied
  - Wenn man nur am Ende einfügt / entfernt, sind Felder **viel** schneller als Listen !
  - **Grund 1:** Bei der Liste müssen wir für jede Operation **vier** Zeiger umbiegen, das kostet Zeit

    Das wiegt deutlich schwerer als das gelegentlich notwendige Reallozieren beim dynamischen Feld
  - **Grund 2:** Bei der Liste müssen wir für jedes Element einzeln Speicher allo**z**ieren, beim dynamischen Feld tun wir das für viele Element auf einmal

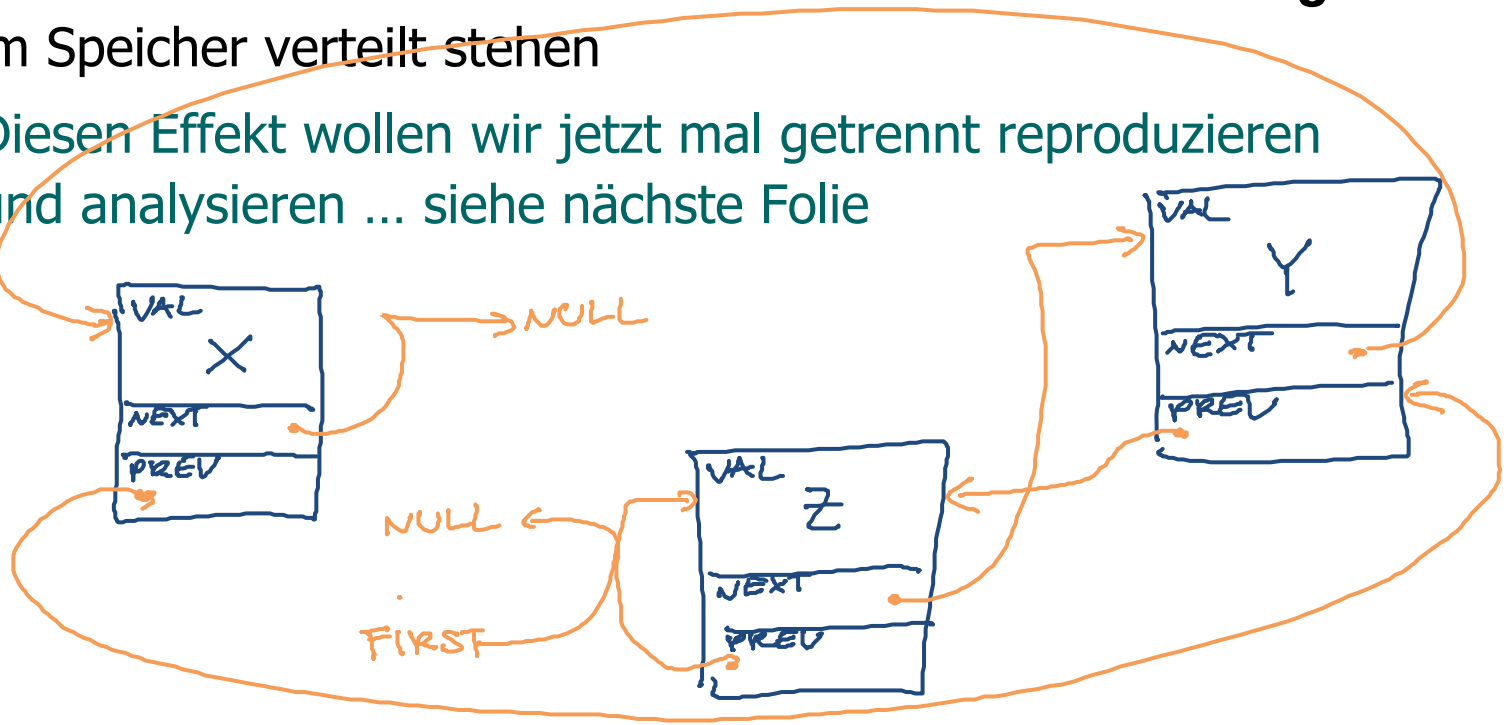
    Der allo**z**ierte Speicher ist zwar am Ende der**s**elbe, aber Speicherallo**k**ation hat einen fixen Overhead pro Aufruf

Laufzeit Listen vs. Felder 4/4

UNI FREIBURG

- Es gibt noch einen Grund
  - Die sogenannte **Lokalität** der Speicherzugriffe
  - Bei einem Feld stehen ja, wie gesagt, die n Elemente im Speicher hintereinander
  - Bei einer verketteten Liste können die Elemente **beliebig** im Speicher verteilt stehen

    Diesen Effekt wollen wir jetzt mal getrennt reproduzieren und analysieren … siehe nächste Folie



Lokalität Speicherzugriffe

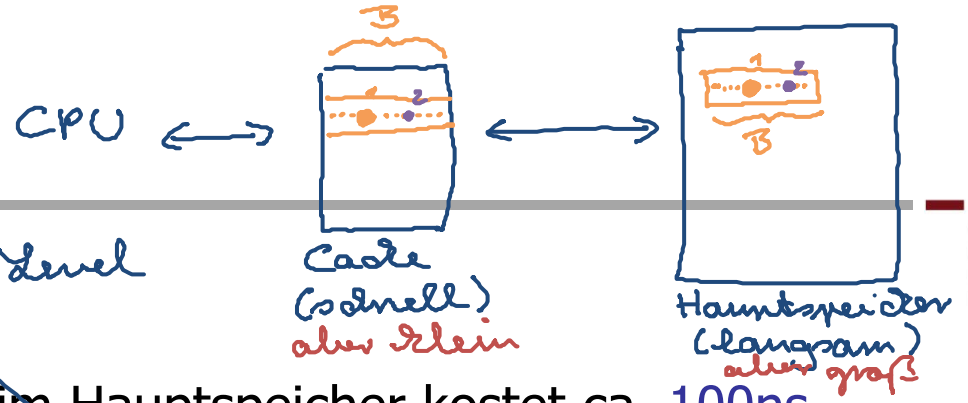
order1 : 1, 2, 3, 4, 5
order2 : 2, 5, 3, 1, 4

UNI FREIBURG

- Einfach(s)tes Beispiel
  - Wir addieren die n Elemente eines Feldes auf
    - … in der natürlichen Reihenfolge: 1 + 2 + 3 + 4 + 5
    - … in einer zufälligen Reihenfolge: 2 + 5 + 3 + 1 + 4
  - Das Ergebnis ist in beiden Fällen **identisch**
  - Die Anzahl der Operationen ist ebenfalls **identisch**
  - Was ist mit der Laufzeit?

Laufzeit für zufällige Reihenfolge
bis zu 20 mal langsamer und mehr
(für n = 500M uns)

CPU Cache



- Prinzip / Aufbau
  - Zugriff auf ein Byte im Hauptspeicher kostet ca. 100ns
  - Zugriff auf ein Byte im (L1-)Cache kostet ca. 1ns
  - Bei Zugriff auf ein oder mehrere Bytes im Hauptspeicher holt man gerade einen ganzen Block von ~ 100 Bytes in den Cache
  - Solange dieser Block im Cache ist, braucht man für Bytes aus diesem Block nicht mehr auf den Hauptspeicher zuzugreifen
  - Der Cache hat Platz für viele solcher Blöcke (die cache lines)
    - ein typischer L1-Cache ist ~ 100 Kilobytes groß
  - Ist der Cache voll, wird einer der Blöcke entfernt
    - z.B. der least recently used (LRU) Block
    - das soll aber heute nicht das Thema sein

Disk Cache

UNI FREIBURG

- Prinzip / Aufbau
  - Den Lesekopf einer Festplatte an eine bestimmte Stelle zu bewegen kostet ~ 5ms (seek time)
  - Ist man an einer Stelle kann man mit ~ 100 MB / Sekunde Daten lesen (transfer rate)

    10ms pro Byte
  - Deshalb geht das Betriebssystem wie folgt vor
    - Wird ein Byte von der Platte gelesen, wird gleich ein ganzer Block eingelesen (z.B. 128 KB auf einmal)
    - Solange dieser Block im Speicher ist, braucht man für Bytes aus diesem Block nicht mehr auf die Platte zuzugreifen und spart sich die seek time
    - Ist der Speicher voll, muss man sich wieder überlegen, welche Blöcke man drin behält

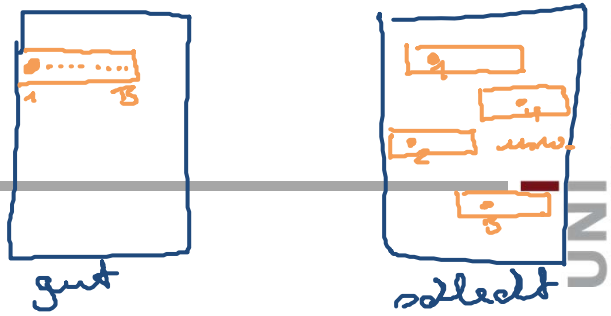
Block-Operationen 1/7

UNI FREIBURG

- Terminologie
  - Wir haben einen langsamen und einen schnellen Speicher
  - Der langsame Speicher ist in Blöcke der Größe B unterteilt
  - Der schnelle Speicher ist M groß (Platz für M/B Blöcke)
  - Stehen die Daten nicht im schnellen Speicher, wird der entsprechende Block in den schnellen Speicher geladen
  - Das Programm kann sich aussuchen, welche Blöcke im schnellen Speicher gehalten werden
  - Wir zählen nur die **Anzahl der Block-Operationen**

    In der Praxis hat man auch noch die Kosten für das Verwalten der Blöcke im schnellen Speicher, insbesondere welcher Block entfernt wird wenn der Speicher voll ist ... das ignorieren wir hier

Block-Operationen 2/7



- Für B Operationen hat man also
  - im besten Fall nur 1 Block-Op. **"gute Lokalität"**
  - im schlechtesten Fall B Block-Op. **"schlechte Lokalität"**
- Die folgenden Variationen ...
  - ... machen nur einen (kleinen) konstanten Faktor in der Anzahl der Block-Operationen aus:
    - die genaue Aufteilung des langsamen Speichers in Blöcke
    - ob die Speichereinheit 1 Byte oder 4 Bytes oder 8 Bytes ist
- Man beachte:
  - Das Ganze wird erst interessant, wenn die Eingabegröße n größer als M ist, sonst passt ja die gesamte Eingabe in den schnellen Speicher und man hat trivial $\lceil n/B \rceil$ Blockoperationen

Block-Operationen 3/7

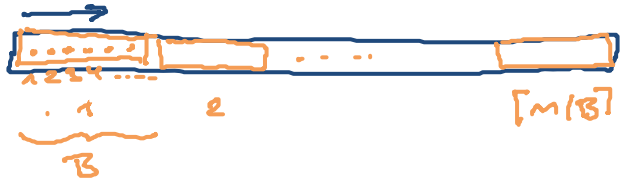
UNI FREIBURG

- Typische Werte (für einen Server)
  - CPU Cache: B = 128 Bytes, M = 6 x 32 KB (L1), 6 x 256 KB (L2)
  - Disk Cache: B = 64 Kilobytes, M = 64 GB

    Die meisten Betriebssysteme benutzen alles, was vom Hauptspeicher gerade nicht genutzt wird, als Disk Cache
  - Sinnvollerweise wählt man B so, dass die transfer time für einen Block ein Bruchteil der seek time ist
- Noch etwas Terminologie
  - Die Block-Operationen beim CPU Cache nennt man cache misses
  - Die Block-Operationen beim Disk Cache nennt man oft IOs

    IO oder I/O = Input/Output
  - Man spricht auch von Cache-Effizienz und IO-Effizienz

Block-Operationen 4/7



- Beispiel 1: Unser ArraySumMain Programm
  - Wenn wir über die n Elemente in der Reihenfolge 1, 2, 3, ... iterieren, ist die Anzahl Block-Operationen: $\lceil n/B \rceil$
  - Wenn wir über die n Elemente in einer zufälligen Reihenfolge iterieren, ist die Anzahl Block-Op. im schlechtesten Fall: n wenn $n >> M$
  - Das ist ein Faktor von B Unterschied und das ist der Hauptgrund für den beobachteten Laufzeitunterschied

    Man beachte, dass auch bei der zufälligen Reihenfolge pro Element auf 4 benachbarte Bytes (ein int) auf einmal zugegriffen wurde

    Außerdem wird, wenn nicht n >> M, das nächste Element manchmal zufällig schon im schnellen Speicher stehen

    Deswegen ist der Unterschied in der Praxis deutlich < B

Block-Operationen 5/7

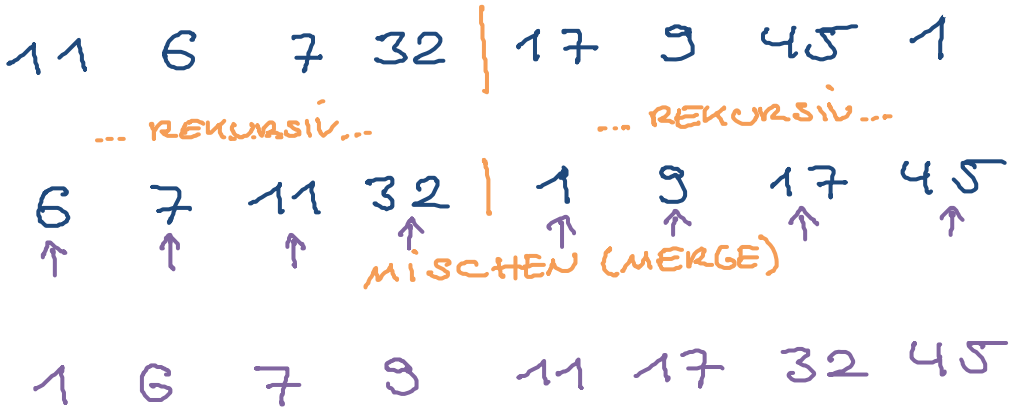
- Beispiel 2: Sortieren mit MergeSort
  - Kurze Wiederholung der Funktionsweise:

    Teile das Feld in zwei gleich große Teile

    Sortiere die beiden Teile rekursiv

    Mische die beiden sortierten Folgen zu einer sortierten Folge

    Für das Ü7 können Sie annehmen, dass n eine Zweierpotenz ist, so dass die beiden Teile immer **exakt** gleich groß sind



Block-Operationen 6/7

UNI FREIBURG

- Beispiel 2: Sortieren mit MergeSort
  - Wiederholung der Analyse der Anzahl **gewöhnlicher** Operationen

$n = \text{Zweierpotenz} = 2^k$, $k = \log_2 n \in \mathbb{N}$

$$T(n) \leq \underbrace{2 \cdot T(n/2)}_{\text{für die Rek.}} + \underbrace{A \cdot n}_{\text{für das Mischen}}$$

$$\leq 2 \cdot [2 \cdot T(n/4) + A \cdot n/2] + A \cdot n$$

$$= 4 \cdot T(n/4) + 2 \cdot A \cdot n$$

$$\vdots$$

$$\leq \underbrace{2^k}_{=n} \cdot \underbrace{T(n/2^k)}_{=1} + \underbrace{k}_{=\log_2 n} \cdot A \cdot n$$

$$= O(n \cdot \log n)$$

Für das Ü7 müssen Sie so eine rekursive Formel für IO(n) = Anzahl Blockoperationen aufstellen und dann auflösen wie oben.

Block-Operationen 7/7

UNI FREIBURG

- Beispiel 2: Sortieren mit MergeSort
  - Analyse der Anzahl Block-Operationen … Ü7, Aufgabe 2
  - Herauskommen soll: $\Theta(n/B \cdot \log_2(n/B))$
  - Es geht sogar: $\Theta(n/B \cdot \log_{M/B}(n/B))$

    Dazu teilt man auf jeder Rekursionsstufe in $k = M/B$ Teile, sortiert die rekursiv und mischt die k sortierten Teilfolgen dann zu einer Folge

    Beim Mischen von k Folgen braucht man zu jedem Zeitpunkt für jede Folge genau einen Block, also $k \cdot B = M$ insgesamt
  - **Lerne:** wenn man das (Kosten-)Modell ändert, kann sich auch ändern, welcher Algorithmus optimal ist

Literatur / Links

UNI FREIBURG

- Cache-Effizienz / IO-Effizienz
  - In Mehlhorn/Sanders:

    2 Introduction 2.2.1 External Memory
  - In Cormen/Leiserson/Rivest

    Nothing! [zero matches for the word "cache"]
  - In Wikipedia

    http://en.wikipedia.org/wiki/Cache

    http://de.wikipedia.org/wiki/Cache

    (da wird das Prinzip eines Caches beschrieben, es gibt aber keinen separaten Artikel zur Cache-Effizienz bei Algorithmen)