**IBN Übungsblatt 6** Tutor: Denitsa Ilieva

Lyubomira Dimitrova, Dung Nguyen, Maryna Charniuk

## Aufgabe 1

Mit der Verwendung einer invertierten Seitentabelle, wenn ein Prozess auf die virtuellen Seitentabelle zugreifen will, kann die Hardware den physischen Seitenrahmen nicht einfach finden, weil sie immer (nicht nur bei den Seitenfehlern) sowohl nach Prozess-ID, als auch nach Seiten-Index durchsuchen muss. Je größer der virtuelle Adressraum ist, desto mehr Zeit braucht die Hardware für das Suchen. In dieser Situation kann ein TLB helfen.

Wenn TLB mit häufig benutzten Seiten voll ist, muss es wieder durch die ganze Tabelle nach Prozess-Id und Seiten-Index gesucht wird. Das wäre dann "worst case".

## Aufgabe 2

**a.** Drei Seiten nach Ende der Anfragen im Speicher geladen sind (nach Strategie):

- **FIFO**: 2, 3, 0
- **LRU**: 2, 3, 0

**b.** Nehmen wir an, im Speicher haben wir 3 Seitenrahmen. Die folgenden Reihenfolge der angefragten Seiten kann die erste Seite als Opfer für Clock und LRU unterschiedlich auswählen: 1,2,3,4,2,5,6

**Clock**

| *Seite* | *1* | *2* | *3* | *4* | *2* | *5* | *6* |
|---|---|---|---|---|---|---|---|
| *1* | 1 \| 1 | 1 \| 1 | → 1 \| 1 | 4 \| 1 | 4 \| 1 | → 4 \| 1 | 4 \| 0 |
| *2* | → | 2 \| 1 | 2 \| 1 | → 2 \| 0 | → 2 \| 1 | 2 \| 0 | **6** \| 1 |
| *3* | | → | 3 \| 1 | 3 \| 0 | 3 \| 0 | 5 \| 1 | → 5 \| 1 |

**LRU**

| *Seite* | *1* | *2* | *3* | *4* | *2* | *5* | *6* |
|---|---|---|---|---|---|---|---|
| *1* | → 1 | → 1 | → 1 | 4 | 4 | → 4 | **6** |
| *2* | | 2 | 2 | → 2 | 2 | 2 | → 2 |
| *3* | | | 3 | 3 | → 3 | 5 | 5 |

Wie oben, wird die Seite 2 bei dem **Clock**-Algorithmus als erstes ausgewählt, wobei bei dem **LRU**-Algorithmus die Seite 4 als erstes.

## Aufgabe 3

**2 Seitenfehler.**

Angenommen das Wort bzw. der Maschinenbefehl befindet sich auf einer Seite (d.h. kann nicht mehrere Seiten umfassen).

Wenn in der Seitentabelle das i/v-Bit des zu ladenden Wortes = invalid ist, haben wir ein Seitenfehler. Bei einem Maschinenbefehl handelt es sich um eine Folge von Bytes, die das CPU steuern. Es kann passieren, dass die Seite mit unserem Machinenbefehl auch nicht im Speicher ist, was einen zweiten Seitenfehler auslöst.

## Aufgabe 4

Zusammenhängend versuchen die beide Algorithmen die Seite, die am seltensten sowohl in der Vergangenheit (`LRU`) als auch zukünftig (`OPT`) benutzt wird, auslagern. Im Prinzip ist zu jedem Zeitpunkt die Seitenfehleranzahl bzw. -rate unterschiedlich. Nehmen wir die folgenden Reihenfolge der angefragten Seiten an:
7, 0, 1, 2, 0, 3, 0, 4, 2, 3.
Angenommen gibt es im Speicher 3 Seitenrahmen. Der Zeiger zeigt an der Seite, die nach dem Algorithmus zunächst ausgelagert wird.

**OPT**

| *Seite* | *7* | *0* | *1* | *2* | *0* | *3* | *0* | *4* | *2* | *3* |
|---|---|---|---|---|---|---|---|---|---|---|
| *1* | → 7 | → 7 | → 7 | 2 | 2 | 2 | 2 | 2 | **2** | 2 |
| *2* | | 0 | 0 | 0 | 0 | 0 | → 0 | → 4 | → **4** | → 4 |
| *3* | | | 1 | → 1 | → 1 | → 3 | 3 | 3 | **3** | 3 |
| *Fehler?* | F | F | F | F | | F | | F | | |

**LRU**

| *Seite* | *7* | *0* | *1* | *2* | *0* | *3* | *0* | *4* | *2* | *3* |
|---|---|---|---|---|---|---|---|---|---|---|
| *1* | → 7 | → 7 | → 7 | 2 | 2 | → 2 | → 2 | → 4 | → **2** | 2 |
| *2* | | 0 | 0 | → 0 | 0 | 0 | 0 | 0 | **0** | 0 |
| *3* | | | 1 | 1 | → 1 | 3 | 3 | 3 | **3** | → 3 |
| *Fehler?* | F | F | F | F | | F | | F | **F** | |

Bis zum Zeitpunkt des Ausruf der Seite *4* ist die Anzahl der Seitenfehler gleich, aber der Zeiger zeigt an den unterschiedlichen Seiten. Das ist der Grund, warum ein Seitenfehler beim LRU-Algorithmus entsteht aber nicht beim OPT-Algorithmus, wenn Seite *2* angefragt wird. Zu diesem Zeitpunkt ist im Speicher beim OPT- noch die Seite *2* vorhanden. Im Allgemein wird der Fall eines Seitenfehlers dadurch festgestellt, wenn die Seite nicht im Speicher gefunden ist. Dann wird die gezeigte Seite durch diese Seite ersetzt. Da die Position des Zeigers von der Reihenfolge der angefragten Seiten abhängt, ist die Fehlerrate jedes Algorithmus unterschiedlich.

## Aufgabe 5

Code und Logs in Moodle hochgeladen :)

## Aufgabe 6

Laut Definition bestehen **Arbeitsbereiche** (*working sets*) aus Menge der Seiten, die in den letzten $\Delta$ Speicherzugriffen benutzt wurden. Wenn diese Seiten gemeinsam benutzt werden können (im Fall von Multiprocessing), können sie gleichzeitig zu verschiedenen Arbeitsbereichen gehören. z.B. wenn viele Nutzer bei einem `Time-Sharing`-System gleichzeitig die gleiche Aufgabe haben, werden die Daten dazu eher geteilt statt für jeden kopiert, dann sind manche Seiten gleichzeitig im Arbeitsbereich jedes Nutzers zugreifbar.

## Aufgabe 7

int $\Delta \leftarrow$ *Eingabe*

Lese Inhalt der Datei mit Dateinamen *name* in ein Array *pageAccesses* ein

*int*[] *pageAccesses* $\leftarrow$ *list_of_pages*

Initialisiere zwei Sets *tmpSet* mit `elements` vom Typ `int` und *workingSets* mit `elements` vom Typ `set(int)`

**if** $\Delta >$ *pageAccesses.length* **then** ▷ `Es gibt nur ein Arbeitsbereich`
  **for** each `element` in *pageAccesses* **do**
    Füge `element` in *tmpSets* ein
    Füge *tmpSet* in *workingSets* ein
  **end for**

**else** ▷ $\Delta \leq$ *pageAcesseses.length*
  int $i \leftarrow \Delta - 1$
  int *count* $\leftarrow 0$
  int *tmp* $\leftarrow 0$

  **do**
    **for** int $x \leftarrow i - \Delta$ to $i$ **do**
      Füge *pageAccesses*[$x$] in *tmpSet* ein
    **end for**

    *count* $\leftarrow$ *tmpSet.length*
      ▷ `Überprüfe ob in` ***workingSets*** `der größte/die größten Arbeitsbereich(en) enthalten ist/sind`
    **if** *tmp* $\leq$ *count* **then**
      **if** *tmp* $<$ *count* **then**
        Empty *workingSets*
        *tmp* $\leftarrow$ *count*
      **end if**

```
        Füge tmpSet in workingSets
      end if

      i ← i + 1
    while i < pageAccesses.length
end if
return workingSets              ▷ der größte/die größten, falls mehr als 1
```

## Aufgabe 8

a) "Rowhammer Problem" ist ein Phänomen, dass in einigen neuen DRAM-Geräten passiert, bei denen durch wiederholten Zugriff auf zwei Speicherorte im virtuellen Adressraum des Prozesses Bit-Flips an einem dritten "Opfer"-Position verursacht werden können. Der Speicherort des Opfers kann sich außerhalb des virtuellen Adressraums des Prozesses befinden, in einer anderer DRAM-Zeile. Das Phänomen ist möglich geworden, weil bei der DRAM-Herstellung die Chips immer verkleinert werden um mehr Speicher zu produzieren, dadurch können DRAM-Zellen elektrisch miteinander interagieren. Bei der wiederholten Zugriffen kann den Wert einer Zelle von 1 auf 0 oder umgekehrt sich ändern. Das passiert in zwei Situationen:

   - **Adressauswahl:** wenn die zwei Adressen **verschiedenen DRAM-Zeilen** in derselben Bank zugeordnet sind, werden die DRAM-Zeilen durch Übertragen (lesen-schreiben) aktiviert, was in den benachbarten Zeilen "bit flips" bei wiederholten Aktivierung verursachen kann. **Eine mögliche Lösung** das zu verhindern: die Adressen müssen auf die gleichen DRAM-Zeilen verweisen. Bei dem ersten Aufruf bleibt die Zeile in Cache und bei dem zweiten Aufruf muss sie nicht nochmal aktiviert werden.
   - **Cache umgehen:** Durch das Leeren des Caches mit den Befehlen `clflush(X)` und `clflush(Y)` werden die Speicherzugriffe an den zugrunde liegenden DRAM gesendet. Das verursacht wiederholendes Aktivieren der Zeilen. **Eine mögliche Lösung** das zu verhindern: keine Verwendung der Befehlen **mfence**, was hilft "bit flips" zu minimieren.

b) Es wird das untere Bit der physischen Seitennummer von dem Seitentabelleneintrag ausgewählt, weil trotz der Veränderung (0 auf 1 oder 1 auf 0) von dem unteren Bit wird es auf die gültigen physikalische Seite gezeigt - es wird eine Seitenzahl erzeugt, die dem physischen Speicher des Systems passt. In anderem Fall, wenn das obere Bit durch "bit flips" verändert wird, ist die Seitenzahl größer als der physische Speicher, was fehlerhaft ist.

c) Das gespeicherte Wort wird im binären System dargestellt, deshalb folgt nach der Manipulation von einem Bit die Änderung der ganzen Zahl.