A. Büscher, J. Mäter,
D. Müller, W. Rupflin

Dortmund, 4. Juni 2007
Besprechung: 03.07. - 05.07.

Übungen zur Vorlesung

Betriebssysteme, Rechnernetze und verteilte Systeme I

Sommersemester 2007

Projekt 2

Bearbeitungshinweise zu Projekt 2:

- Für die Bearbeitung dieses Projekts gelten weiterhin die Hinweise von Projekt 1.
- Die zu ändernden Quelltexte sind entsprechend markiert. In diesem Projekt müssen lediglich die inhalte einiger Methoden geändert werden. Dazu dürfen Sie auch zusätzliche Variablen innerhalb der jeweiligen Klassen definieren.
- Die Abgabe muss bis zum 24.06.2007 um 24 Uhr erfolgt sein.

Bei diesem Projekt sollen Sie die unterschiedlichen Paging-Verfahren, die Sie im Umgang mit virtuellem Arbeitsspeicher kennen gelernt haben, im Umfeld eines gecachten Speicher einsetzen. Das Szenario sieht dabei wie folgt aus: Ein **Worker** benötigt unterschiedliche Daten aus einem **Storage**. Von diesen Daten kennt er die Position also deren Index. Da der **Storage** „langsam“ ist, wird ein **Cache** zwischengeschaltet, der eine begrenzte Menge dieser Daten aufnehmen kann. Dieser **Cache** kann dabei unterschiedliche Strategien einsetzen, um Seiten, die z.Z. im **Cache** sind durch Seiten im **Storage** auszutauschen.

Auf den hinteren Seiten dieser Projektbeschreibung finden Sie die benötigten Quelltexte.

Aufgabe 2.1 Bearbeiten Sie die folgenden beiden allgemeinen Aufgabenstellungen.

(a) Implementieren Sie die noch ausstehenden Cache-Strategien LFU (Least Frequently Used), LRU (Least Recently Used) und LIFO (Last In First Out). Als Beispiel ist die Strategie FIFO (First In First Out) bereits implementiert.

(b) Wieso funktionieren die Cache-Strategien, bei den vorgegebenen Zugriffsmustern gleich gut bzw. schlecht?

Aufgabe 2.2 Beantworten Sie die Frage „Wann ist LFU den anderen Strategien überlegen?“, indem Sie die beiden folgenden Aufgabestellungen bearbeiten.

(a) Welche Eigenschaften müssen die Anfragen haben, damit LFU die beste Strategie darstellt?

(b) Implementieren Sie in der Klasse WorkerLFU eine „zufällige“ Zugriffsreihenfolge, so dass LFU den Cache besser nutzt, als die anderen Strategien.

Listing 1: Worker.java

```
import java.util.Random;

/**
 * Diese Klasse dient dem Testen ihrer Implementierung. Bei Durchgang 1 sollten
 * die implementierten Verfahren bei 1% Trefferquote im Cache liegen und bei
 * Durchgang 2 bei 99,9%.
 *
 * Sie duerfen diese Klasse nicht veraendern.
 */
public class Worker {

    public static void main(String[] args) {
        int storagesize = 1000;
        int cachesize = 10;

        Storage storage = new Storage(storagesize);

        System.out.println("Durchgang 1:");
        testStrategy1(storage, cachesize, new StrategyLIFO(cachesize,
                storagesize));
        testStrategy1(storage, cachesize, new StrategyLFU(cachesize,
                storagesize));
        testStrategy1(storage, cachesize,
                new StrategyFIFO(cachesize, storagesize));
        testStrategy1(storage, cachesize, new StrategyLRU(cachesize,
                storagesize));

        System.out.println();
        System.out.println("Durchgang 2:");
        testStrategy2(storage, cachesize, new StrategyLIFO(cachesize,
                storagesize));
        testStrategy2(storage, cachesize, new StrategyLFU(cachesize,
                storagesize));
        testStrategy2(storage, cachesize,
                new StrategyFIFO(cachesize, storagesize));
        testStrategy2(storage, cachesize, new StrategyLRU(cachesize,
                storagesize));

        WorkerLFU.main(args);
    }

    private static void testStrategy1(Storage storage, int cachesize,
            CacheStrategy cs) {
        Cache cache = new Cache(storage, cachesize, cs);
        Random rng = new Random();
        int max = storage.getSize();
        for (int i = 0; i < storage.getSize() * 1000; i++) {
            int index = rng.nextInt(max);
            int[] data = cache.getElement(index);
            if (data[0] != index) {
                System.err.println("Falsche Daten erhalten.");
            }
        }
        cache.printStats();
    }
```

```
    private static void testStrategy2(Storage storage, int cachesize,
            CacheStrategy cs) {
        Cache cache = new Cache(storage, cachesize, cs);

        for (int i = 0; i < storage.getSize() * 1000; i++) {
            int index = i / 1000;
            int[] data = cache.getElement(index);
            if (data[0] != index) {
                System.err.println("Falsche Daten erhalten.");
            }
        }

        cache.printStats();
    }
}
```

Listing 2: Storage.java

```
/**
 * Diese Klasse enthaelt den "langsamen", großen Speicher aller Daten.
 *
 * Diese Klasse darf nicht veraendert werden.
 */
public class Storage {

    private int[][] data = null;

    /** Construktor
     * @param size Groesse des Speichers
     */
    public Storage(int size) {
        data = new int[size][1];
        for (int x = 0; x < size; x++) {
            data[x][0] = x;
        }
    }

    /** Lade Daten von einer bestimmten Stelle des Speichers.
     *
     * @param i Position der Daten
     * @return geforderte Daten
     */
    public int[] getData(int i) {
        return data[i];
    }

    /**
     * Liefere die Groesse des Speichers.
     * @return Speichergroesse
     */
    public int getSize() {
        return data.length;
    }
}
```

Listing 3: WorkerLFU.java

```
import java.util.Random;

/**
 * Diese Klasse testet die verschiedenen Strategien. Die Methode testStrategyLFU
 * soll so implementiert werden, dass die Stragie LFU den anderen Verfahren
 * überlegen ist. Die angeforderten Seiten sollen dabei zufallsbasiert
 * ausgewaehlt werden, wobei diese Zufaelligkeit etwas modifiziert werden darf.
 * Die anderen Methoden sollen nicht veraendert werden.
 */
public class WorkerLFU {

    public static void main(String[] args) {
        int storagesize = 1000;
        int cachesize = 10;

        Storage storage = new Storage(storagesize);

        System.out.println();
        System.out.println("Durchgang LFU:");
        testStrategyLFU(storage, cachesize, new StrategyLIFO(cachesize,
                storagesize));
        testStrategyLFU(storage, cachesize, new StrategyLFU(cachesize,
                storagesize));
        testStrategyLFU(storage, cachesize, new StrategyFIFO(cachesize,
                storagesize));
        testStrategyLFU(storage, cachesize, new StrategyLRU(cachesize,
                storagesize));
    }

    private static void testStrategyLFU(Storage storage, int cachesize,
            CacheStrategy cs) {
        Cache cache = new Cache(storage, cachesize, cs);

        Random rng = new Random();

        int max = storage.getSize();
        for (int i = 0; i < storage.getSize() * 1000; i++) {
            int index = rng.nextInt(max);
            int[] data = cache.getElement(index);
            if (data[0] != index) {
                System.err.println("Falsche Daten erhalten.");
            }
        }

        cache.printStats();
    }
}
```

Listing 4: Cache.java

```
import java.text.*;

/**
 * Diese Klasse enthaelt den "schnellen", kleinen Speicher aller Daten.
 * Diese Klasse darf nicht veraendert werden.
 */
public class Cache {
    /** Speicher hinter dem Cache */
    private final Storage    storage;
    /** Strategie des Caches */
    private final CacheStrategy strategy;
    /** aktuelle gecachte Daten */
    private int[][]          data       = null;
    /** Positionen der gecachten Daten im Storage */
    private int[]            indizes    = null;
    // Statistik
    /** Anzahl der Anforderungen eines bestimmten Speicherbereichs */
    private int[]            requests   = null;
    /** Summe aller Anforderungen */
    private int              requestcount = 0;
    /** Anzahl der Treffer im Cache */
    private int              hitcount   = 0;

    /**
     * Construktor
     * @param storage Speicher hinter dem Cache
     * @param size Groesse des Caches
     * @param cs Strategie des Caches
     */
    public Cache(Storage storage, int size, CacheStrategy cs) {
        this.storage = storage;
        this.strategy = cs;
        data = new int[size][0];
        indizes = new int[size];
        requests = new int[storage.getSize()];
        for (int x = 0; x < indizes.length; x++) {
            indizes[x] = -1;
            requests[x] = 0;
        }
    }

    /**
     * Liefert das Element mit index i aus dem Storage. Wenn sich dieses Element
     * im Cache befindet wird kein Zugriff auf den Storage durchgefuert. Sonst
     * wird mittels der CacheStrategy ein CacheElement ausgewaehlt, dass durch
     * die angeforderte Seite ersetzt wird.
     * @param i Index der angeforderten Daten
     * @return angeforderte Daten
     */
    int[] getElement(int i) {
        requests[i]++;
        requestcount++;
        strategy.notifyRequest(i, indizes, requests);
        for (int x = 0; x < indizes.length; x++) {
            if (indizes[x] == i) {
```

```
                hitcount++;
                return data[x];
            }
        }
        int index = strategy.replaceIndex(i, indizes, requests);
        indizes[index] = i;
        data[index] = storage.getData(i);
        return data[index];
    }

    /**
     * Ausgabe der Statistiken.
     */
    public void printStats() {
        double ratio = ((double) hitcount) / ((double) requestcount) * 100.0;
        System.out.println("Cache: " + NumberFormat.getInstance().format(ratio) + "%");
    }
}
```

Listing 5: StrategyLIFO.java

```
public class StrategyLIFO extends CacheStrategy {

    StrategyLIFO(int cachesize, int storagesize) {
        super(cachesize, storagesize);
    }

    public void notifyRequest(int index, int[] indizes, int[] requests) {
    }

    public int replaceIndex(int frame, int[] indizes, int[] requests) {
        return 0;
    }
}
```

Listing 6: StrategyLRU.java

```
public class StrategyLRU extends CacheStrategy {

    StrategyLRU(int cachesize, int storagesize) {
        super(cachesize, storagesize);
    }

    public void notifyRequest(int index, int[] indizes, int[] requests) {
    }

    public int replaceIndex(int frame, int[] indizes, int[] requests) {
        return 0;
    }
}
```

Listing 7: CacheStrategy.java

```
/**
 * Dieses Interface definiert die allgemeinen Methoden fuer die
 * Ersetzungsstrategien.
 *
 * Diese Klasse darf nicht veraendert werden.
 */
public abstract class CacheStrategy {

    /** Groesse des Caches */
    protected final int cachesize;

    /** Groesse des Speichers */
    protected final int storagesize;

    /**
     * Construktor
     * @param cachesize Groesse des Caches
     * @param storagesize Groesse des Speichers
     */
    public CacheStrategy(int cachesize, int storagesize) {
        this.cachesize = cachesize;
        this.storagesize = storagesize;
    }

    /**
     * Eine bestimmte Seite wurde angefordert
     * @param index
     * @param indizes
     * @param requests
     */
    public abstract void notifyRequest(int index, int[] indizes, int[] requests);

    /**
     * Bestimmt den Index im Cache, der durch die angeforderte Seite ersetzt
     * werden soll.
     *
     * @param frame index der angeforderten Seite
     * @param indizes indizes der Seiten im Cache
     * @param requests Anzahl der requests jeder Seite
     * @return Index der zu ersetzenden Stelle im Cache
     */
    public abstract int replaceIndex(int frame, int[] indizes, int[] requests);
}
```

Listing 8: StrategyFIFO.java

```
/**
 * Diese Klasse enthaelt ein Beispiel fuer eine Strategie nach dem
 * FIFO-Verfahren. Diese Klasse darf nicht veraendert werden.
 */
final class StrategyFIFO extends CacheStrategy {

    private int index = 0;

    StrategyFIFO(int cachesize, int storagesize) {
        super(cachesize, storagesize);
    }

    public void notifyRequest(int i, int[] indizes, int[] requests) {
        // Nothing to do
    }

    public int replaceIndex(int frame, int[] indizes, int[] requests) {
        // Solange noch ein Platz frei ist, wird dieser benutzt.
        for (int x = 0; x < cachesize; x++) {
            if (indizes[x] < 0) {
                return x;
            }
        }

        // Zu entfernenden Index merken
        int next = index;
        // Naechsten Index berechnen
        index = (index + 1) % cachesize;

        return next;
    }
}
```

Listing 9: StrategyLFU.java

```
public class StrategyLFU extends CacheStrategy {

    StrategyLFU(int cachesize, int storagesize) {
        super(cachesize, storagesize);
    }

    public void notifyRequest(int index, int[] indizes, int[] requests) {
    }

    public int replaceIndex(int frame, int[] indizes, int[] requests) {
        return 0;
    }
}
```