University of Arizona, Department of Computer Science

CSc 340 Foundations of Computer Systems — Assignment 5

Christian Collberg

Due Tuesday April 17, at 11:59pm

1 Overview

High-level languages such as Java are typically implemented in C for efficiency. For example, parts of the Java standard library are written in C.

Java also has facilities that allow users to call out to C. The idea is to allow you to code the performance-critical part of your program in C and then call that part from your main Java program.

In this assignment you will write a hash table module `hash.c` in C, a Java class `Hashtable.java` that interfaces to `hash.c`, and then compare the efficiency of this implementation to that of `java.util.Hashtable`.

2 Assignment

The purpose of this assignment is to implement this Java class:

```
public class Hashtable {
   static {System.loadLibrary("hash");}
   public static final int OPEN = 1;
   public static final int LINEAR = 2;
   public static final int QUADRATIC = 3;
   public static final int DOUBLE = 4;

   private long ht;

   public Hashtable(int size, int algorithm) {ht = NEW(size,algorithm);}
   public Hashtable(int size) {ht = NEW(size,OPEN); }
   public Hashtable() {ht = NEW(101,OPEN);}

   private native long NEW(int size, int algorithm);
   private native void insert(long ht, String key, String value);
   private native String lookup(long ht, String key);
   private native boolean member(long ht, String key);
   private native void delete(long ht, String key);
   private native void print(long ht);

   public void insert(String key, String value) {insert(ht, key, value);}
   public String lookup(String key) {return lookup(ht, key);}
   public boolean member(String key) {return member(ht, key);}
   public void delete(String key) {delete(ht, key);}
   public void print() {print(ht);}
}
```

The `native` methods are implemented in a C module called `hash.c`. The statement ⌜`System.loadLibrary("hash")`⌝ makes sure this library is loaded by the Java interpreter.

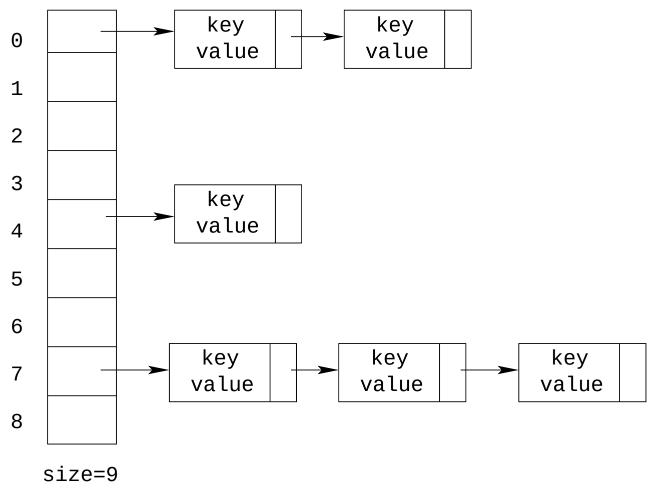
As we will see, the hash table implementation represents the hash table as a pointer to a struct. In this Java interface the variable `ht` holds this pointer.

The file `hash.h` describes the functions that `hash.c` needs to implement:

```
typedef struct node {
  struct node * next;
  char* key;
  char* value;
} *NODE;

typedef struct hash_table {
  NODE* table;
  int size;
} *HASHTABLE;

enum hash_algorithm {hash_open,hash_linear,hash_quadratic,hash_double};

extern HASHTABLE hash_new(int size, int algorithm);
extern void hash_insert(HASHTABLE t, const char* key, const char* value);
extern char* hash_lookup(HASHTABLE t, const char* key);
extern int hash_member(HASHTABLE t, const char* key);
extern void hash_delete(HASHTABLE t, const char* key);
extern void hash_print(HASHTABLE t);
```

3 Algorithm

The hash table has the following structure:



That is, the table consists of an array of pointers to linked lists of nodes. These lists are called *buckets*. Each node holds a pair of strings, the `key` and the `value`. Intially the table is empty (every entry is `NULL`). When we want to insert a new $\langle \textbf{KEY}, \textbf{VALUE} \rangle$ pair we compute $H(\textbf{KEY})$, the *hash value* of **KEY**. We then search the $H(\textbf{KEY})$ mod `size`:th list for a node whose `key` value is **KEY**. If **KEY** is found we replace its value with **VALUE**. Otherwise, we create a new $\langle \textbf{KEY}, \textbf{VALUE} \rangle$-node and insert it first in the bucket.

The `lookup` function performs a similar search, but returns `NULL` if **KEY** is not found.

The hash function $H(x)$ can be very simple. We might, for example, simply add up the ASCII values of the characters in the string. A more sophisticated function might treat the string as an array of 4 or 8-byte chunks, and compute the hash value as the `xor` of these chunks.[1]

4 The Java Native Interface

`JNI` is the part of the Java specification that describes how Java programs call functions in other languages, and vice versa. There is lots of information on the web on how to use `JNI`:

```
http://java.sun.com/products/jdk/1.2/docs/guide/jni
http://ringlord.com/platform/java/jni-howto.html
```

For more links, search for ⌜`Java JNI`⌝ at `google.com`.

For this assignment you don't really have to worry too much about how to use `JNI` (unless you do some of the extensions). Everything has been prepared for you; just type `make` and *The Right Thing*™ will happen.

The interface between `hash.c` (where you write all your code) and `Hashtable.java` is given in `Hashtable.c`:

```
#include <jni.h>
#include "Hashtable.h"
#include <stdio.h>
#include "hash.h"

JNIEXPORT jlong JNICALL Java_Hashtable_NEW
  (JNIEnv *env, jobject obj, jint size, jint algorithm) {
    return (jlong) hash_new(size,algorithm);
}

JNIEXPORT void JNICALL Java_Hashtable_insert
  (JNIEnv *env, jobject obj, jlong ht, jstring key, jstring value) {
  const char *KEY = (*env)->GetStringUTFChars(env, key, 0);
  const char *VALUE = (*env)->GetStringUTFChars(env, value, 0);
  hash_insert((HASHTABLE)ht, KEY, VALUE);
  (*env)->ReleaseStringUTFChars(env, key, KEY);
  (*env)->ReleaseStringUTFChars(env, value, VALUE);
}

JNIEXPORT jstring JNICALL Java_Hashtable_lookup
  (JNIEnv *env, jobject obj, jlong ht, jstring key) {
  char *VALUE;
  const char *KEY = (*env)->GetStringUTFChars(env, key, 0);
```

[1] If you do this, be careful not to go outside the string (where there may be extra characters lurking) or you might experience random behavior.

```
  VALUE = hash_lookup((HASHTABLE)ht, KEY);
  (*env)->ReleaseStringUTFChars(env, key, KEY);
  return (*env)->NewStringUTF(env, VALUE);
}

JNIEXPORT jboolean JNICALL Java_Hashtable_member
  (JNIEnv *env, jobject obj, jlong ht, jstring key) {
  const char *KEY = (*env)->GetStringUTFChars(env, key, 0);
  int res = hash_member((HASHTABLE)ht, KEY);
  (*env)->ReleaseStringUTFChars(env, key, KEY);
  if (res)
     return JNI_TRUE;
  else
     return JNI_FALSE;
}

JNIEXPORT void JNICALL Java_Hashtable_delete
  (JNIEnv *env, jobject obj, jlong ht, jstring key) {
  const char *KEY = (*env)->GetStringUTFChars(env, key, 0);
  hash_delete((HASHTABLE)ht, KEY);
  (*env)->ReleaseStringUTFChars(env, key, KEY);
}

JNIEXPORT void JNICALL Java_Hashtable_print
  (JNIEnv *env, jobject obj, jlong ht) {
  hash_print((HASHTABLE)ht);
}
```

This module uses special functions (`GetStringUTFChars`, `NewStringUTF`, ...) defined by `JNI` to convert between Java's data types and C's data types. C strings, for example, are null-terminated arrays of bytes, whereas Java's strings consist of a length and an array of shorts. Whenever we pass a string from C to Java or from Java to C we have to perform the appropriate conversion.

5 The makefile

Copy the files from `/home/cs340/spring01/prog5` to a new directory and type `make`. The following should happen:

```
javac Hashtable.java
javah -jni Hashtable
cc -G -I/usr/j2se/include/ -I/usr/j2se/include/solaris Hashtable.c hash.c -o libhash.so
Hashtable.c:
hash.c:
javac Test.java
cc -g -o Test hash.c Test.c
hash.c:
Test.c:
javac Timing.java
```

You can now run the test programs, which, at this point, don't do much:

```
java Test
java Timing
Test
```

The `makefile` encodes the following relationships between the different files:

```
Hashtable.java    --javac-->  Hashtable.class  --javah -jni-->  Hashtable.h

Hashtable.h
Hashtable.c       --cc-->     libhash.so
hash.h
hash.c

Test.c
hash.h            --cc-->     Test
hash.c

Test.java         --javac-->  Test.class
Hashtable.class
```

I.e. `javah` will look at the signatures of the native methods (encoded in `Hashtable.class`) and construct C function definitions from these (in `Hashtable.h`). `Hashtable.h`, `Hashtable.c`, `hash.h`, and `hash.c` are compiled together and linked into a *dynamic library*, `libhash.so`. This is what ⌜`System.loadLibrary("hash")`⌝ will load on the fly when Java executes `Hashtable.java`.

These relationships are encoded in the `makefile`:

```
FILES = Hashtable.class Hashtable.h libhash.so Test.class Test Timing.class

INCLUDE = -I/usr/j2se/include/ -I/usr/j2se/include/solaris/
CC = cc
SHARED = -G

all: ${FILES}

Hashtable.class: Hashtable.java
        javac Hashtable.java

Test.class: Test.java
        javac Test.java

Timing.class: Timing.java
        javac Timing.java

Hashtable.h: Hashtable.class
        javah -jni Hashtable

libhash.so: Hashtable.c hash.c hash.h
        ${CC} ${SHARED} ${INCLUDE} Hashtable.c hash.c -o libhash.so

Test: Test.c hash.c hash.h
        ${CC} -g -o Test hash.c Test.c
```

6 Evaluation

You should write a Java program `Timing.java` that times your hash table implementation and `java.util.Hashtable` on some reasonable and random input. Discuss your findings in a file `TIMING-RESULTS`. Use `System.currentTimeMillis()` to time the program. You should time both successful searches (when what you're looking for is actually in the table) and unsuccessful ones.

7 Extension 1 [5% extra credit]

Keep the hash buckets sorted by `key`. Modify your algorithm to speed up unsuccessful searches. How does this affect your timing results?

8 Extension 2 [10% extra credit]

When a hash table gets too full it can be advantageous to expand it. Typically, we keep track of the average length of the hash buckets and on every insertion into the table check if the length is greater than some threshold, for example 2. If it is, we create a new hash table (often twice the size of the original one) and rehash the elements into this new table. The old table is then discarded.

Implement such a rehashing policy. How does this new policy affect your timing results?

9 Extension 3 [15% extra credit]

The hashing algorithm described above is known as *open hashing.* *Closed hashing* is a different class of algorithm, where there are no linked hash buckets. Rather, the hash table is simply an array of $\langle \mathbf{KEY}, \mathbf{VALUE} \rangle$-pairs. When a collision occurs (i.e. when an element hashes to the same array index as another element already in the table) the new element is *forwarded* to a new location. This process is continued until an empty entry is found, which is where we insert the new element.

In general, on the i:th probe into the table, we check to see if the table entry number

$$f(i) = (H(x) + g(i)) \bmod \texttt{size}$$

is empty. $H(x)$ is the hash function and $g(i)$ is defined differently for different algorithms:

Linear Probing This is the simplest algorithm. We simply check each entry following the original probe until an empty one is found. I.e. $g(i) = i$.

Quadratic Residue In this algorithm $g(i) = \langle 0, 1^2, -(1^2), 2^2, -(2^2), 3^2, -(3^2), \cdots, i^2, -(i^2), \cdots \rangle$.

Double Hashing Both linear and quadratic probing have the problem that elements start "lumping" together as the table gets full. Double hashing solves this problem by introducing a second hash function, H_2, such that $f(i) = (H_1(x) + i * H_2(x)) \bmod \texttt{size}$. There are some side conditions that need to be fulfilled in order to make this algorithm work correctly.[2]. The easiest thing to do is to make sure that the table size is a double prime (i.e. both `size` and `size`-2 are prime) and that

$$f(i) = (H(x) \bmod \texttt{size} + i * (H(x) \bmod (\texttt{size} - 2)) \bmod \texttt{size}.$$

Implement these different strategies. Be careful to handle deletion correctly. You get 5% per implementation.

10 Extension 4 [10% extra credit]

Augment the `Hashtable.java` API such that it implements `java.util.Iterator`.

11 Extension 5 [10% extra credit]

Java has very primitive facilities for timing programs. On unix the system call `getrusage()` returns information about the running program. Write a Java class and a corresponding C module that together give Java programs access to the timing information returned by `getrusage()`.

This is an open-ended extension – you should design your own Java timing API, keeping in mind what facilities will be useful to a Java programmer. Document the API fully, provide test cases, and make sure that the `Makefile` builds everything correctly.

See ⌜`man getrusage`⌝, `/usr/include/sys/resource.h` and `/usr/include/sys/time.h` to get started.

12 Honors Section

You should do at least extension 1 for *no* extra credit.

13 Turnin

When you have completed the program, submit your files by typing make turnin. You may turn in as many times as you want; `turnin` will always replace the previously turned-in version with the new version.

If you have implemented any of the extensions you should describe them in the README file, so that the graders know what to test for.

This is an individual assignment. Don't show your code to anyone, don't read anyone's code, don't discuss the details of the code with anyone. If you need help with the assignment see the TA or the instructor.

[2]See, for example, `http://www.eece.unm.edu/faculty/heileman/hash/node4.html`