Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Fall Term 2014

Systems@ETH zürich

SYSTEMS PROGRAMMING AND COMPUTER ARCHITECTURE
Assignment 4: Assembly

Assigned on: **16th Oct 2014**
Due by: **23rd Oct 2014**

NOTE: Unless otherwise stated, the assembly code in this assignment is x86-64 assembly.

1 Implementing a hash table

To practice the different C concepts you have learned so far, in this question we ask you to implement a hash table which maps strings to arbitrary data pointers (`void*` in C). This is illustrated in the figure below.

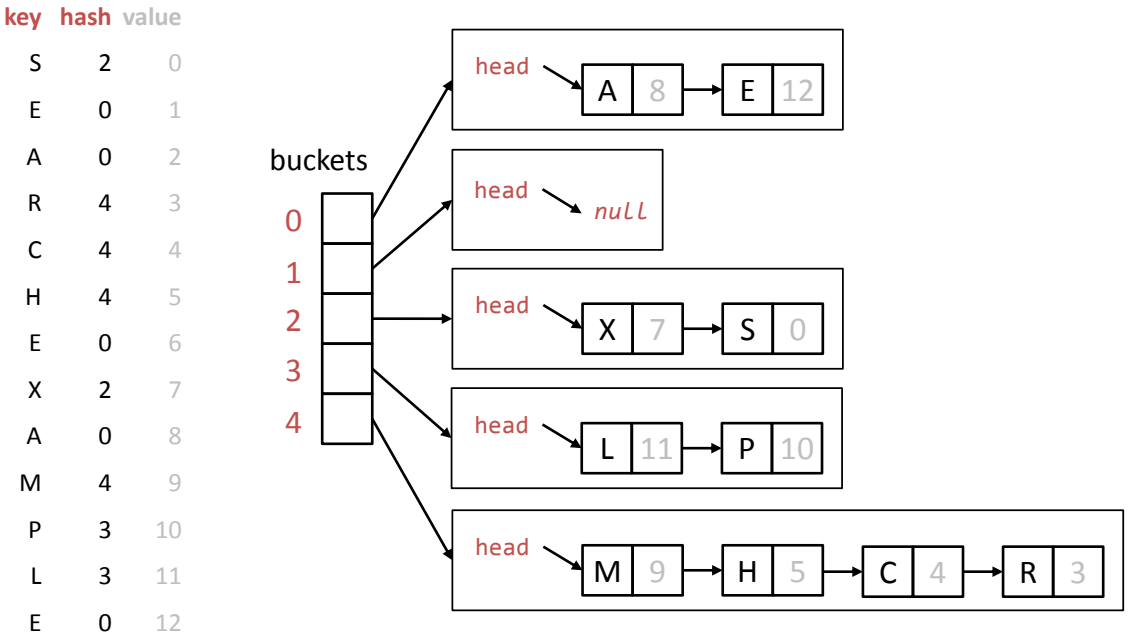

Figure 1: Hashing with separate chaining

On the course website there is a header file `<hash_table.h>` which defines the interface you need to implement. The data structures are already defined and you should implement a few basic operations:

- `hash_create`: Allocates the table structure and an array for the buckets.
- `hash_release`: Releases the entire hash table, including all its entries properly.
- `hash_insert`: Inserts a key/value pair into the hash-table.
- `hash_find`: Finds an element in the hash-table and returns the corresponding data value or `NULL` if the key does not exist.
- `hash_delete`: Deletes an element in the hash-table. This function should also call the registered function to free the data.

Note that the search keys are strings and each key should map to one of the hash buckets. For this, compute a *hash function* which transforms the search key into an array index – i.e. assuming we have N slots this should return an integer in the range $[0, N)$. To deal with collisions, where multiple keys hash to the same bucket, you should chain the elements in a linked list.

2 Assembly basics

2.1 Array basics

For each of the arrays declared below provide:

a) the size of one element in bytes,

b) the total size of the array in bytes,

c) the byte address of element i if the array starts at address $x_{<arrayindentifier>}$,

d) two different C expressions for accessing element i of the array.

e) a C expression that dereferences an actual char, short, or int, at index 2 (index (2,0), index (2,0,0) respectively) of the array (i.e. char value = A[2]).

```
char    A[5];
char   *B[3];
char  **C[8];
short   D[4];
short  *E[9];
int     F[4];
int    *G[7];
```

2.2 Addressing modes

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x204	0xFF
0x208	0xCD
0x20C	0x21
0x210	0x11

Register	Value
%rax	0x2
%rcx	0x204
%rdx	0x3

Fill in the following table showing the types (i.e., immediate, register, memory) and the values of the indicated operands:

Operand	Type	Memory address	Value
%rax			
0x210			
$0x210			
(%rcx)			
4(%rcx)			
5(%rcx, %rdx)			
519(%rdx, %rax)			
0x204(, %rax, 4)			
(%rcx, %rax, 2)			

2.3 Arithmetic operations

Use the values of the memory addresses and registers from Question 1. Handle the different instructions independently. The result of one of the instructions does not affect the others. Fill in the following table showing the effects of the following instructions, both in terms of the register or memory location that will be updated and the resulting value:

Instruction	Destination	Computation and result
`addl %eax, (%rcx)`		
`subl %edx, 4(%rcx)`		
`imull (%rcx, %rax, 4), %eax`		
`incl 8(%rcx)`		
`decl %eax`		
`subl %edx, %ecx`		

2.4 leal and movl

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x108	0xFF
0x10C	0xCD
0x110	0x21

Register	Value
%rax	0x100
%rcx	0x4
%rdx	0x1

What is the difference between the two instructions? What value ends up in %ecx? Write the formula!

```
movl 8(%rax, %rdx, 4), %ecx
```

```
leal 8(%rax, %rdx, 4), %ecx
```

2.5 Condition codes

Consider the instruction `addl` %rax, %rbx. As a side-effect, it sets the condition flags (OF, SF, ZF, CF) according to the result.

Assuming a 4-bit machine, convert the given decimal pairs `(a, b)` to their binary representation and perform the addition. Give both the arithmetical value and the interpreted value (2's complement) of the result. List the condition flags that are set.

(-1, -1), (+4, TMin), (TMax, TMax), (TMax, -TMax),
(TMin, TMax), (TMin, TMin), (-1, TMax), (2, 3).

2.6 Reading Condition Codes with C

In this exercise you will obtain and print the processor's condition codes for different assembly instructions using a C program. To facilitate your task we have prepared a program skeleton that already does most of the work (ccodes.c). You can download the skeleton (`ccodes.c`) from the course web page.

On Intel processors the condition codes are stored in the 64-bit wide RFLAGS register. The program skeleton first executes an assembly instruction and stores the resulting contents of the RFLAGS register to a variable. **Your task is to complete the function getccodes()** (no other part of the program needs to be modified). This function extracts the four condition codes of interest (sign flag, carry flag, zero flag, overflow flag) from the RFLAGS register and stores their values to a C struct of type `struct ccodes`.

The layout of the RFLAGS register is described in the Intel Architecture Software Developer's Manual (Volume 1, Section 3.4.3); the link to the Intel manuals is indicated on the course web page.

When the program is complete, compile and run it to test if it functions properly. For example, the command may look like this: `gcc -Wall -Wextra ccodes.c -o ccodes`. You can also add new test cases to the `main()` function.

ADVICE: All required tools to compile the programs should be installed on the lab machines. If you want to build it on your own machine, make sure to install gcc and gcc-multilib (e.g. on Ubuntu: `sudo apt-get install build-essential gcc-multilib`).

Hand In Instructions

Except for Question 2.6, this is a paper exercise. If you want your solution to be revised please hand it in during your exercise class on the due date. Upload your ccodes.c (Question 2.6) to a subfolder **assignment4** of your SVN folder. Refer to Assignment 1 for instructions on using SVN.