Project: Symbol Lookup

Introduction

One of the core activities in debugging crashes is looking at stack traces, or activation records.

When working with compiled languages such as C and C++, these typically come in the form of a list of memory addresses from the code when it was executing.

An example from a crash running on an iPhone:

```
Last Exception Backtrace:
0   CoreFoundation                      0x01cca1c8 0x1c43000 + 553416
1   libobjc.A.dylib                     0x018b18e5 0x18ac000 + 22757
2   CoreFoundation                      0x01cc9fbb 0x1c43000 + 552891
3   CRExample-simulator                 0x00002e0e 0x1000 + 7694
4   CRExample-simulator                 0x00002d66 0x1000 + 7526
5   CRExample-simulator                 0x00002f65 0x1000 + 8037
6   libobjc.A.dylib                     0x018c3880 0x18ac000 + 96384
7   UIKit                               0x0036db39 0x355000 + 101177
8   UIKit                               0x0036dac5 0x355000 + 101061
9   UIKit                               0x0046e8d1 0x355000 + 1153233
10  UIKit                               0x0046ecc6 0x355000 + 1154246
...
```

In the above you can see the frame index, the associated module, and the instruction pointer of the running code. The instruction pointer is then split out into the base address of the function being executed, and the offset into the compiled code for that function.

(With some additional information, that offset can be used to determine the line of source code that it corresponds to, but that is beyond the scope of this exercise.)

As great as hex addresses are, what would be really nice is to see the names of the functions.

Typically, a compiler generates a file of debugging information along with the executable binary, that can be used to lookup those function names, and more.

However, if we don't have access to that file, we can still determine the function names by using the symbol table stored within the binary.

Your task

The symbol table will contain the names of functions, along with the address of their first instruction (usually with some additional information.)

You will be provided with a dump of a symbol table, in text format, containing a mapping of function names to starting addresses.

The first line of this file is a special case - it contains the last valid address of code within the program, on a line by itself.

Subsequent lines contain a function name, and a hexadecimal address, separated by a comma, one per line.

You need to write a program that will accept this file, and a list of addresses (in hexadecimal.)

It should output, one per line - the address, followed by the name of the function it belongs to. If the address is out of range (that is, it comes before the address of the first function in the table, or after the last valid address) - it should print the character "!" instead of a name.

Example Symbol Table Dump File

```
0x000000002d3d16c0
_CFDictionaryGetValue,0x000000002d3c7b30
_CFBasicHashFindBucket,0x000000002d3c7bb8
_CFRetain,0x000000002d3c8ab4
__CFRetain,0x000000002d3c8b20
__CFSetTSD,0x000000002d3c8c60
_CFAllocatorAllocate,0x000000002d3c8cf0
__CFGetTSD,0x000000002d3c8d60
_CFDataCreateWithBytesNoCopy,0x000000002d3c8de0
__CFRuntimeCreateInstance,0x000000002d3c8e28
_CFPropertyListCreateFromXMLData,0x000000002d3c9038
```

Example Usage

```
$ ./lookup_symbols symbol_dump.txt 0x2d3c8cf9 0x3c000
0x2d3c8cf9 _CFAllocatorAllocate
0x3c000 !
$
```

Assumptions

1. The first valid address is the lowest address encountered in the dump file, not 0x0.
2. Functions are stored contiguously in memory, with no gaps.

Program invocation

Your program should be able to be invoked in the following manner:

programname path_to_dump_file address1 address2 adress3 [...]

i.e.

```
$ ./lookup.py symbol_table.txt 0x2525 0x3325 0x33523
```

Submission Guidelines

You may write your submission in any language.

If you choose to use a compiled language, your submission should include a simple Makefile that will produce a runnable program.

Your submission should include a readme.txt file telling us what compiler was used, or in the case of an interpreted language such as Python, Ruby, Perl, etc - which version of the interpreter is required to run your code, and any other dependencies.

Please send us your submission as a zip archive, or gzipped tarball (tgz.)