1 Project 4

Due: Nov 18, by 11:59p.

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. It then hints at how these requirements can be met. Finally, it describes exactly what needs to be submitted.

1.1 Aims

The aims of this project are as follows:

- To make you manipulate code as data.
- To expose you to dynamic memory allocation and function pointers in C.
- To make you build `Makefile`'s which make use of non-trivial linking concepts.

1.2 Requirements

Implement a module which implements the abstract data-type (ADT) specified by fn-trace.h which provides information (address, code-length and # of ingoing and outgoing calls) for all functions called directly or indirectly by a specified function in a dynamically loaded module:

```c
/** Information associated with a function. */
typedef struct {
  void *address;          /** start address of function */
  unsigned length;        /** # of bytes taken by function code. */
  unsigned nInCalls;      /** # of direct calls to this function */
  unsigned nOutCalls;     /** # of direct function calls in function body */
} FnInfo;

/** FnsData is an opaque struct FnsData which can be implemented as
 *   desired by an implementation. This struct will hold a collection
 *   of FnInfo's.
 */
typedef struct FnsData FnsData;
```

```c
/** Return pointer to opaque data structure containing collection of
 *  FnInfo's for functions which are callable directly or indirectly
 *  from the function whose address is rootFn.
 */
const FnsData *new_fns_data(void *rootFn);

/** Free all resources occupied by fnsData. fnsData must have been
 *  returned by new_fns_data().  It is not ok to use to fnsData after
 *  this call.
 */
void free_fns_data(FnsData *fnsData);

/** Iterate over all FnInfo's in fnsData.  Make initial call with NULL
 *  lastFnInfo.  Keep calling with return value as lastFnInfo, until
 *  next_fn_info() returns NULL.  Values must be returned sorted in
 *  increasing order by fnInfoP->address.
 *
 *  Sample iteration:
 *
 *  for (FnInfo *fnInfoP = next_fn_info(fnsData, NULL); fnInfoP != NULL;
 *       fnInfoP = next_fn_info(fnsData, fnInfoP)) {
 *    //body of iteration
 *  }
 *
 */
const FnInfo *next_fn_info(const FnsData *fnData, const FnInfo *lastFn-
Info);
```

You may assume the following about the code being traced:

- A function makes function calls using the `CALL` instruction (op-code byte `0xE8`) followed by 4 bytes giving the offset to the called function (in little-endian order) relative to the start of the **next** instruction.
- A function is terminated using a `RET` instruction (op-code bytes `0xC3`, `0xCB`, `0xC2` or `0xCA`).

Submit a `submit/prj4-sol` folder in your `i220`X repository in github such that typing `make` within that folder produces a `fn-trace` executable within that directory. When the executable is run, it must start execution at the main program provided (see below) which drives your implementation of the above API.

Your program must meet the following additional requirements:

- It should not have any hard-coded limit on the maximum number of functions which can be traced; the limit should only be set by the amount of available memory.

- The program should not treat as instructions bytes within immediate data which have the same encoding as the encodings for `CALL` and `RET` instructions. (Given the reliable x86-64 instruction-length decoding provided by the `x86-64_lde` module (see below), this should not be a major issue).
- All code must be compiled for the `x86-64` architecture. This is the default with the compilers installed on remote.cs.

1.3 The main Program

You are being provided with a main program to exercise your implementation of the above tracing API. When this program is run, it expects an optional flag `-r`, followed by the following 2 arguments:

MODULE The path to a dynamically-linked module.

FN_NAME The name of a `extern` function in *MODULE* which takes no arguments and returns an `int`.

The program initially loads the specified *MODULE* using the `dlopen()` API (not covered in this course, briefly discussed at the end of Ch. 7 of the text).

It then prints out the result of calling *FN_NAME* with no arguments.

Finally, it uses your implementation of the ADT described above, to provide information for all the functions called directly or indirectly by *FN_NAME*. Specifically, for each such function, it prints out the address of the function (relative address if the `-r` flag is specified), the number of direct calls made to/by that function and the number of bytes in the code for the function.

The program **must** be invoked with the `LD_LIBRARY_PATH` environmental variable containing the `$HOME/cs220/lib` directory.

A log of the execution is available in fn-trace.LOG.

1.4 x86-64 Instruction Length Decoder

This project requires you to scan a sequence of x86-64 instructions and identify selected instructions (with op-codes provided in the **Requirements** section). To perform the scan, you will need to step from one instruction to the next sequential instruction, which requires knowing the length (number of bytes) of the first instruction. A BeaEngine disassembler library has been installed in the course lib directory to assist in this task. To make this even easier, you are being provided with an implementation of the following x86-64_lde.h specification file.

```c
/** Return length of x86-64 instruction pointed to by p.  Returns < 0
 *  on error.
 */
int get_op_length(void *p);
```

1.5 The lib Directory

The course lib directory contains the following 64-bit libraries:

`libbeaengine` The library mentioned above for disassembling x86-64 instructions.

`libcs220` A trivial library which provides help with memory allocation and error reporting:

- Provides checked versions `mallocChk()`, `reallocChk()`, and `callocChk()` of the memory allocation routines which wrap the standard memory allocation routines with the program exiting on failure. The specification file is in memalloc.h.
- Provides routines for reporting errors using `printf()`-style format strings with one modification: if the format string ends with `:`, then `strerror(errno)` is appended to the error-message. The specification file is in errors.h.

You may assume that the environment in which your program will be compiled and run will have these libraries available.

1.6 Provided Files

The prj4-sol directory contains the following files:

main.c This provides the `main()` function described above. The file documents it's external dependencies which must be setup correctly in order to build and run the program.

Instruction Length-Decoder for x86-64 This provides the simple wrapper function around the BeaEngine disassembler library which allows accessing the length of a x86-64 instruction. The specification file is in x86-64_lde.h and and implementation in x86-64_lde.c.

The implementation x86-64_lde.c documents it's external dependencies which must be setup correctly in order to build and run the program.

Function Trace Files The fn-trace.h file contains the specification for the API you are required to implement. The fn-trace.c file contains a skeleton implementation; you should flesh out the function definitions given there (adding in any auxiliary declarations or definitions which may be needed).

do-fn-trace A trivial wrapper shell script which invokes the `fn-trace` program with the `LD_LIBRARY_PATH` environment set up to point to the course `lib` directory.

Makefile A makefile for the project.

The aux-files directory provides the following files:

Test Code Generator A ruby script in gen-fns.rb which generates random functions which can be traced for test purposes. The script is invoked with 2 arguments:

NUM_FNS The number of mutually recursive functions to be generated.

FN_NAME The name of a top-level function to be generated which invokes the mutually recursive functions and returns an `int`. This is the root function which will start the function trace.

The script will write the generated code to standard output; usually, this would be redirected to a file.

The script ensures that the immediate data generated as part of the generated code will contain random bytes with values equal to the op-codes for the `RET` and `CALL` instructions. This ensures that if the program succeeds tracing this test data, then the program does not get confused by such situations.

fns.c Sample test data built by invoking the above script as:

```
$./gen-fns.rb 20 testFn >fns.c
```

Makefile Used to build the shared object module `fns` which serves as input to the `fn-trace` program, as well as a standalone program `fnsTest` with a conditionally-included `main()` function from fns.c.

fn-trace.LOG A sample execution log. Note that the printed function addresses can vary from run to run, but the relative addresses should be the same.

Executable `fn-trace` An executable which implements the project requirements.

1.7 Hints

The following steps are not prescriptive in that you may choose to ignore them as long as you meet all project requirements.

1. Review the examples covered in class dealing abstract data types and dynamic memory allocation.

2. Make sure your local `cs220` repository is up-to-date:

```
# assumes you have set up your account as directed
$ cd ~/cs220
$ git pull
```

3. Copy over the provided files to your `work` folder in your `i220X` local repository:

```
# also assumes you have set up your account as directed
$ cd ~/i220X/work  #X will be a or b
$ cp -pr ~/cs220/projects/prj4/prj4-sol .
$ cd prj4-sol
```

4. Fill in the README template.

5. Using the `x86-64_lde` module, have the `new_fns_data()` function merely print out the length of each instruction in the root function being traced. Your code should be setup to terminate when any `RET` opcode is seen.

6. Look at how you can implement the `FnsData` structure which needs to contain a collection of `FnInfo`'s. There are many different methods of implementing collections, but since the specification requires that elements of the collection be returned sorted and C provides a `qsort()` function which can sort arrays, the most convenient way of implementing the collection of `FnInfo`'s is by using a dynamically grown array of `FnInfo`'s. The techniques discussed in class for building such arrays using `realloc()` can be used:

   - Track the next index to be allocated in the dynamic array as well as the current size of the dynamic array.
   - When adding a new element, if the index is greater-than-or-equal-to the size, `realloc()` the array (possibly doubling the size).
   - Start off with everything at 0 (the size and index at 0, the dynamic array base pointer pointing to `NULL`). That ensures that even inserting a single element causes a `realloc()`.
   - Instead of using naked `realloc()`'s, use the `reallocChk()` provided by `libcs220`.

   The functionality provided by your `FnsData` structure should allow checking to see if there is a `FnInfo` for a particular address as well as adding in a new `FnInfo`.

7. Set up `new_fns_data()` and `free_fns_data()` to allocate/free all memory needed for your implementation of `FnsData`.

8. After your `FnsData` has been initialized in `new_fns_data()`, start tracing the root function. You can start tracing a function as follows:

   - At the start of tracing a function, enter its address into the `FnsData` structure.

   - Process each instruction of the traced function accumulating the total length of the function.

   - If a `CALL` instruction is encountered, check to see if the absolute address of the called function has been entered into the `FnsData` structure. If yes, update the `nInCalls` counter for the called function and keep accumulating the length of the current function being traced. If not, then recursively trace the called function. When that recursive trace returns continue accumulating the current function.

9. Implement the `nextFnInfo()` iterator. At this point, the iterator results will not be ordered. However, you can link with the provided main program and test.

10. Review the `qsort()` function. Use it to sort the dynamic array of `FnInfo`'s by function address.

11. Iterate until you meet all requirements.

More details as to how you may proceed:

- `FnsData` represents a collection of `FnInfo`'s:

  1. The data-structure representing the collection must be initialized.

  2. It should be possible to add a `FnInfo` to the collection.

  3. Given the address of a function, it should be possible to check whether that function is in the collection.

  As pointed out above, a dynamic vector along the lines of what was covered in class seems best suited.

- `new_fns_data()` should probably only be a simple wrapper function which merely initializes the data-structure chosen for `FnsData` and then calls an auxiliary function (say `fn_trace()`) passing it the incoming `rootFn` argument as well as the initialized data-structure.

- The specification for the auxiliary function `fn_trace(void *addr, FnsData fnsData)` is that it will add information about the function represented by `addr` as well as all functions called directly or indirectly by that function to the fnsData collection as long as those functions have not been seen earlier.

- If you ensure that `fn_trace()` is only called for a new function, then `fn_trace()` can start things off by adding the function to the `FnsData` collection (with a known address but unknown length) at some index in the dynamic vector, saving the index in a local variable. Since the entry to `fn_trace()` represents a call to the new function, the `nInCalls` counter for the new function should be initialized to 1.

- An `unsigned char *` pointer can be initialized to the function's start address `addr`. This pointer can be used to scan the instructions for the function specified by `addr`.

- The scan would loop through instructions, incrementing the pointer by the length of each encountered instruction, getting the length by using the provided `get_op_length()` from the `x86-64_lde` module. The loop could also accumulate the length of the function in a suitable variable.

- The loop would terminate when the pointer is pointing to a byte which is an op-code for one of the return instructions.

- If during the loop, the pointer is pointing to the op-code for a call instruction, then the code should get the offset operand for the call. It can do so by setting an `int` pointer to point to the **byte** after the op-code and then dereferencing that pointer (this takes care of endian issues). The address of the function being called will be this offset plus the address of the **next** instruction after the call instruction.

  If the address being called is already in `fnsData`, then nothing needs to be done except increment the `nInCalls` counter for the previously seen function. Otherwise, `fn_trace()` should be called recursively to trace that function. On return from the recursive call, the trace should automatically resume after the `call` instruction.

- When the scan terminates, it should enter the accumulated length into the `FnInfo` at the previously saved index in `fnsData` as well as set `nOutCalls` to the number of call instructions seen during the scan of the function body.

Since manipulating and understanding object and assembly language code is one of the aims of this course, it is perfectly ok to disassemble the provided `fn-trace` executable to get some idea as to how to write your function tracing code.

1.8 Submission

When you are happy with your project, move it over from your work directory to your submit directory:

```
$ cd ~/i220X #X is either a or b
$ git mv work/prj4-sol submit
```

```
$ git commit -a -m 'suitable comment'
$ git push
```

This should submit your project as `submit/prj4-sol` via github. Your submission should not include any object files or executables; this will be prevented by the provided *.gitignore* file.

If submitting late, please drop an email to the TA for your section:

Section A yli241@binghamton.edu

Section B rrausha1@binghamton.edu