Objectives

The objectives of the programming assignment are 1) to review your basic C/C++ programming skills 2) to practice using C structs 3) to practice basic C pointer operations 4) to explore how C structs are actually implemented, via assembly language

Background

Structs and Field Offsets

Your program for this project will work with an array of structs '**library'** that are defined in a C program. Each member of the array is a struct that contains data about an e-book. The data type for that struct is declared with:

```
#define AUTHOR_LEN   40
#define TITLE_LEN    20

typedef struct book {
   char author[AUTHOR_LEN + 1];    // first author
   char title[TITLE_LEN + 1];
   unsigned int year;              // year of e-book release
} books;
```

In memory, each struct book occupies a contiguous block of memory. However, it is not necessarily the case that each field of a struct is placed right after the previous one. The reason is that the C compiler sometimes adds padding between fields of a struct to align each field to its natural boundary. This can be observed through the gdb debugger. For example, let us define a simple 2-field struct rec, and then declare an instance of the struct, as follows:

```
struct rec {
   char field1[5];
   int field2;
};

struct rec myRec = {"Hi", 47};        // A global instance of a struct rec
```

If we examined this structure in gdb:

```
(gdb) print myRec
$1 = {field1 = "Hi\000\000\000",
  field2 = 47}

(gdb) print &myRec
$2 = (struct rec *) 0x804a3a0

(gdb) print &myRec.field1
$3 = (char (*)[5]) 0x804a3a0
```

```
(gdb) print &myRec.field2
$4 = (int) 0x804a3a8
```

Notice that field2 starts 8 bytes after the start of field1, not just 5 as you might expect.
Using this kind of exploration with gdb, we can discover the actual offsets of all the fields of any struct, and use that information to then define constants for our assembly code, thus:

```
; Offsets for fields in struct rec.
;
%define FIELD1_OFFSET 0
%define FIELD2_OFFSET 8
```

For this project, you will add the following to your assembly code (in **bookcmp.asm**), replacing the "??" with actual numbers you got from exploring the struct books much as we did with the struct rec in the example above. On gdb, to determine address references, the variable name for struct books will be **library**. This is defined in driver.c and loaded into memory.

```
; Offsets for fields in struct book.
;
%define TITLE_OFFSET ??
%define AUTHOR_OFFSET ??
%define YEAR_OFFSET ??
```

Once we have discovered and defined the offset values, we can use them in our assembly code to access the individual fields in any struct of that type. For example, if the RSI register holds the address of an instance of struct book, then [RSI+YEAR_OFFSET] can be used to reference the year field of the struct. Similarly, [RSI+RCX+TITLE_OFFSET] can be used to access the i-th character of the title string where the value of i is stored in RCX.

Quick Sort

In this project (**proj5.c**), you will be utilizing the qsort (Quick Sort) function defined in **stdlib.h**. The algorithm is described at the wikipedia link. You are not required to understand the operation/algorithm of qsort. However, understanding the details helps you build a good foundation for a future algorithms course.

For implementation details, on your GL terminal, type

```
man qsort
```

In short the quick sort function **sorts** an array of elements based on a comparator operation
The argument list for qsort in the code provided to you, is incomplete. You will have to define your argument names appropriately in place of '??'. Use the function call to 'sort_books' within 'driver.c' as a clue to the order of the arguments and the type of arguments passed.

sort_books(library,numBooks); //Observe parameter ordering here, and use this as a clue to specify the argument list for sort_books

Assignment

You will be writing a mixed-language program (**proj5.c)**, mostly implemented in the C language, that sorts an array of struct books. It will implement the qsort function but will call out to a small subroutine to compare each pair of books, which you will write in assembly language.

Your program will be executed with the help of a driver code (driver.c) which will supply the test case inputs from 10 different test cases named test1.dat to test10.dat. Observe the code provided in driver.c and use it to your advantage. Do not modify the contents of driver.c.

The class blackboard has been updated with a Project 5 directory. This directory already contains the skeleton of your project along with a Makefile that can build and test your project and tell you if it seems to be operating correctly.

Inside the skeleton project you will find the following important files:

- driver.c: Invokes your sort_books function with the test case inputs loaded into memory
- proj5.c: A starting point for creating your project. Includes the definition for sort_books
- Makefile: Describes to the "make" command how to build the C and assembly code described above and link everything together into the proj4 executable
- expected.txt: Contains the output that a correct program should print when stimulated using the provided driver.asm
- test.sh: Shell script for testing your code. This script needs to be made executed before you run make test
- test1.dat to test10.dat - Ten test case inputs, which you can open in any text file viewer to observe its contents. Each test case contains information about a selection of book titles, respective author names and year of publication.

To build your project (from within the provided project4 directory):

make

to test your project first type (Do this only once for a specific project folder)

chmod 777 ./test.sh

then type

make test

The records in the library file contain the information for test input books which

The "driver.c" program's first task will be to process the input, filling in an array of struct book structs in memory. The array will be dimensioned to hold a maximum of 50 books. The driver program will read in a line at a time from a test case file, each line comprising a record of a single book. It will continue reading records until either (a) it hits the end-of-file, or (b) it fills the entire array, whichever comes first. Here are a few sample records from the file:

Breaking Point, Pamela Clare, 2011
Vow, Kim Carpenter, 2012
1491, Charles C. Mann, 2006
Three Weeks with My Brother, Nicholas Sparks, 2004

(Note that the fields in the records are not in the same order as the fields in your struct--that should not matter.) You may also examine, but NOT edit the contents of a test case (test1.dat to test10.dat) file using any text viewer/editor.

Each of the string fields (title, author) in your struct must be null-terminated--that is the reason each is dimensioned as the length limit + 1. So, AUTHOR_LEN, for example, is the actual maximum number of real (i.e., non-null) characters that can be in the author's name, including spaces.

Every record is "clean", meaning all fields are present and of the right format. The only place where a comma (',') appears is as a field separator (i.e., it does not appear embedded in a title, author name, etc.).

Once all of the book records have been read in by driver.c into a books structure variable 'library', it calls sort_books, a function you are required to complete, based primarily on year of publication, oldest year first. For all books published in the same year, you will subsort by title, in alphabetic order (don't be scared--see the Hints section). See the partial sample output below.

You will sort the book titles in the array of structs by implementing qsort, as described in the Background section above. The qsort function will invoke bookcmp which is made available via an external reference to the assembly function you are required to complete. The reference is achieved by mean of a function pointer. A function pointer, also called a subroutine pointer or procedure pointer, is a pointer that points to a function. As opposed to referencing a data value, a function pointer points to executable code within memory. The first argument to qsort will be the array of structures which has been loaded into memory via driver.c (**library**). This array of structures will simultaneously store the final sorted list of books (by title). The qsort function will evaluate a pair of books structures at a time utilizing the bookcmp code which will be implemented in assembly.

You will implement bookcmp in assembly code, in the file bookcmp.asm. It will look in the registers RDI and RSI to determine the addresses (pointers) to a pair of book structures that need to be compared. It will return one of the integer values -1, 0, or +1 in the register EAX, depending on whether book1 is strictly less than, equal to, or greater than book2, respectively. However, there is a catch, you are required to first determine if the books were published in the same **year,** if they were, then you should proceed to compare the book titles lexicographically. If

not, they should be treated as lexicographically equivalent. This initial check must precede the comparison operations.

You will obviously have to implement your C code and assembly code in separate files. The driver.c code will call the function sort_books in the C file proj5.c, and your assembly file bookcmp.asm will be invoked via qsort. The entry point of your assembly subroutine must be called bookcmp. The qsort function in proj5.c must call this bookcmp subroutine. One of your first instructions in bookcmp should be to save the RDI and RSI registers which hold the addresses to a pair of books structures passed in by the qsort function. You can use indexed addressing modes on these.

mov eax, dword [rdi + YEAR_OFFSET] for the first book's year offset passed by the qsort function.

At the end of your program, after the qsort function, you will print out all of the book titles in sorted order, each title terminated by a newline character.

When you build your program, you must compile or assemble each source file separately:

```
linux2% nasm -f elf64 -g -F dwarf bookcmp.asm
linux2% gcc -g driver.c proj5.c bookcmp.o  -o proj5
```

(A Makefile template is provided for this project in the zip file)

Then, you can run the executable **proj5** file produced. A sample run should look like:

```
linux2% ./proj5 > output.txt
```

Breaking Point
Vow
1491
Three Weeks with My Brother
The 10 Habits of Happy Mothers
We Need to Talk About Kevin
=========
Dead until Dark
The Prague Cemetery
We Need to Talk About Kevin
Black Dahlia & White Rose
Glad Tidings
The Handmaids Tale
=========
…
More output follows for all test cases

Note: The graders will use the same set of test files to test your program, provided with this project spec.

Notes/Hints

An important part of this project is using an appropriate indexed addressing mode in your assembly code to access the fields in the struct. Think this through carefully. A clean and logical approach to this problem will yield clean and logical code that is easier to construct and, more importantly, easier to debug.

When subsorting by title, the strings should be compared using dictionary ordering. For example, any string starting with the letter 'a' comes before any string that starts with 'b' (regardless of length). Capital letters should come before any lowercase letters (this is the way the ASCII code works). In the case that one string is a prefix of another, the shorter string come first. E.g., "egg" comes before "eggs".

Note that the strings in struct book are all C-style NULL-terminated strings. You will have to watch for this as you loop to compare strings for subsorting by title. One nice feature is if one title ends before the other, it's null will always be less than any real character in the other title (think about it), so the comparison will work without any special treatment.

You may find the LEA instruction useful. (LEA = Load Effective Address.) The LEA instruction stores what would have been the address in the source operand into the destination operand. For example, lea rdi, [RSI+TITLE_OFFSET]

will move RSI + TITLE_OFFSET and store it in RDI.

In order for code in other files (like proj5.c) to call your assembly subroutine, you must declare the bookcmp label to be global. (If the label is not global, then only code from the same file can use that label.) Thus, you must include the following declaration in your bookcmp.asm file:

```
global bookcmp
```

The global declarations are typically made just after the SECTION .text declaration.
Also, to tell nasm that book1 and book2 are defined elsewhere, you must have the following declaration in your assembly language program:

```
extern book1, book2
```

This will allow you to use book1 and book2 as labels for the memory locations that hold pointers to the two books to be compared.
Conversely, in order to be able to call the subroutine bookcmp from your C code, you must insert the following declaration in your C code to tell the gcc compiler that bookcmp is defined in a different file. Otherwise, it will only look for bookcmp in the current file (and not find it). You must also specify details of how it expects arguments, and what it returns. To tell gcc this, include the declaration: extern int bookcmp(void);

What to Submit

Before you submit your program, you should test your program sorting the data files that we provided.

Use the UNIX submit command on the GL system to turn in your project. You should submit **seven** files: (1) the C source code that implements the sorting, in a file named proj5.c, (2) your assembly language code impementing the bookcmp function, named as bookcmp.asm, (3) the test shel script provided to you (unmodified), (4) the driver code provided to you, unmodified, (5) the header file that implements the array of structures (6) you Readme documentation and (7) a typescript file in the even your code does not produce all expected test case results.

The UNIX command to do this should look something like:

```
submit CMSC313_Deepak proj5 proj5.c bookcmp.asm test.sh driver.c book.h Readme typescript
```