CSCI 4210 — Operating Systems
Homework 1 (document version 1.2)
Dynamic Memory Allocation, Pointer Arithmetic, and Files

- This homework is due in Submitty by 11:59PM EST on **(v1.2)** Thursday, June 8, 2023
- You can use at most three late days on this assignment
- This homework is to be done individually, so **do not share your code with anyone else**
- **(v1.2)** Place your code in `hw1.c` for submission; you may also optionally include your own header files
- You **must** use C for this assignment, and all submitted code **must** successfully compile via `gcc` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; we will also use `-Werror`, which will treat all warnings as critical errors
- All submitted code **must** successfully compile and run on Submitty, which currently uses Ubuntu v20.04.6 LTS and `gcc` version 9.4.0 (`Ubuntu 9.4.0-1ubuntu1~20.04.1`)

Hints and reminders

To succeed in this course, do **not** rely on program output to show whether your code is correct. Consistently allocate **exactly** the number of bytes you need regardless of whether you use static or dynamic memory allocation. Further, deallocate dynamically allocated memory via `free()` at the earliest possible point in your code.

Make use of `valgrind` to check for errors with dynamic memory allocation and dynamic memory usage. As another helpful hint, close open file descriptors (or `FILE` pointers) as soon as you are done using them.

Finally, always read (and re-read!) the `man` pages for library functions, system calls, etc.

Homework specifications

In this first homework, you will use C to implement a rudimentary cache of words, which will be populated with strings read from an input file. Your cache must be a dynamically allocated hash table of a given fixed size that handles collisions by simply replacing the existing word.

This hash table is really just a one-dimensional array of `char *` pointers. These pointers should all initially be set to `NULL`, then set to point to dynamically allocated strings for each cached word.

No square brackets allowed!

To emphasize and master the use of pointers and pointer arithmetic, **you are not allowed to use square brackets** anywhere in your code! As with our first lecture exercise, if a `'['` or `']'` character is detected, including within comments, that line of code will be removed before running `gcc`. (Ouch!)

To detect square brackets, consider using the command-line `grep` tool as shown below.

```
bash$ grep '\[' hw1.c
...
bash$ grep '\]' hw1.c
...
```

Can you combine this into one `grep` call? As a hint, check out the `man` page for `grep`.

Review the posted code examples to better understand pointer arithmetic. In brief, square bracket expressions can generally be rewritten using pointer arithmetic by removing the square brackets, enclosing the sum of the array variable and the index in parentheses, then dereferencing the resulting pointer. A few equivalent examples follow:

```
str[32] = 'A';
*(str+32) = 'A';

values[i] += 20;
*(values+i) += 20;

results[j] = j * 3.14;
*(result+j) = j * 3.14;

if ( strcmp( a, &b[10] ) == 0 ) { ... }
if ( strcmp( a, &(*(b+10)) ) == 0 ) { ... }
if ( strcmp( a, b+10 ) == 0 ) { ... }
```

Note that in this last example, we do not need to dereference the pointer since we are then using the address-of `&` operator; combined, these two operations negate one another.

Command-line arguments and memory allocation

The first command-line argument specifies the size of the cache, which therefore indicates the size of the dynamically allocated `char *` array that you must create. Use `calloc()` to create this array of "placeholder" pointers. And use `atoi()` (or `strtol()`) to convert from a string to an integer on the command line.

Next, your program must open and read the regular file specified by the second command-line argument. Your program must parse and extract all words (if any) from the given file. Here, a word is any string of two or more alpha characters (see below). If a collision occurs, replace the pre-existing entry.

Initially, your cache is empty, meaning it is an array of `NULL` pointers. Storing each valid word therefore also requires dynamic memory allocation. For this, use `calloc()` if the cache array slot is empty; otherwise, to replace an existing value, use `realloc()` if the size of the required memory differs from what is already allocated.

For words (e.g., `"arch"`), be sure to calculate the number of bytes to allocate as the length of the given word plus one, since strings in C are implemented as `char` arrays that end with a `'\0'` character.

Note that you are **not** allowed to use `malloc()` anywhere in your code!

Is it a valid word—and how do you "hash" it?

For this assignment, words are defined as containing only alpha characters (see `isalpha()`) and consisting of at least two characters in length. All other characters therefore serve as delimiters. And note that words are case sensitive (e.g., `Lion` is different than `lion`).

(v1.1) You can assume that the maximum valid word length is 128 bytes.

To determine the cache array index for a given word, i.e., to properly "hash" the word, write a separate function called `hash()` that calculates the sum of each ASCII character in the given word as an `int` variable, then applies the "mod" operator to determine the remainder after dividing by the cache array size.

As an example, the valid word `Meme` consists of four ASCII characters, which sum to $77 + 101 + 109 + 101 = 388$. If the cache array size was 17, for example, then the array index for `Meme` would be the remainder of 388/17 or 14.

Required Output

When you execute your program, you must display a line of output for each valid word that you encounter in the given file. For each word, display the cache array index and whether you called `calloc()` or `realloc()`—or did not need to change the already existing memory allocation.

Given the `lion.txt` example file, you could run your code as follows:

```
bash$ ./a.out 17 lion.txt
```

Below is sample output from the above program execution that shows the format you must follow:

```
Word "Once" ==> 15 [calloc]
Word "when" ==> 9 [calloc]
Word "Lion" ==> 11 [calloc]
Word "was" ==> 8 [calloc]
Word "asleep" ==> 5 [calloc]
Word "little" ==> 8 [realloc]
Word "Mouse" ==> 11 [realloc]
Word "began" ==> 16 [calloc]
Word "running" ==> 4 [calloc]
Word "up" ==> 8 [realloc]
Word "and" ==> 1 [calloc]
Word "down" ==> 15 [nop]
Word "upon" ==> 8 [realloc]
Word "him" ==> 12 [calloc]
...
```

Further, when you have finished processing the input file, show the contents of the cache by displaying a line of output for each non-empty entry in the cache. Use the following format:

```
Cache:
index 0 ==> "on"
index 1 ==> "gnawed"
index 2 ==> "King"
index 3 ==> "LITTLE"
index 4 ==> "went"
index 5 ==> "PROVE"
index 6 ==> "to"
index 7 ==> "tree"
index 8 ==> "little"
index 9 ==> "said"
index 10 ==> "MAY"
index 11 ==> "Mouse"
index 12 ==> "him"
index 13 ==> "FRIENDS"
index 14 ==> "GREAT"
index 15 ==> "the"
index 16 ==> "began"
```

Error handling

If improper command-line arguments are given, report an error message to `stderr` and abort further program execution. In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then aborting further program execution. Only use `perror()` if the given library or system call sets the global `errno` variable.

Error messages must be one line only and use the following format:

```
ERROR: <error-text-here>
```

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty.

Note that this assignment will be available on Submitty a minimum of three days before the due date. Please do not ask when Submitty will be available, as you should first perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, make use of the `DEBUG_MODE` technique to make sure that Submitty does not execute any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of q is %d\n", q );
    printf( "here12\n" );
    printf( "why is my program crashing here?!\n" );
    printf( "aaaaaaaaaaaaagggggggghhhh square brackets!\n" );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE hw1.c
```

Second, output to standard output (`stdout`) is buffered. To disable buffered output for grading on Submitty, use `setvbuf()` as follows:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure good results on Submitty, this is a good technique to use.