CSCI 4210 — Operating Systems
Homework 1 (document version 1.1)
Dynamic Memory Allocation, Pointer Arithmetic, and Files

- This homework is due in Submitty by 11:59PM on **(v1.1)** Thursday, January 23, 2025
- You can use at most three late days on this assignment
- This homework is to be done individually, so **please do not share your code with others**
- Place your code in `hw1.h` and `hw1.c` for submission; you may also 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 v22.04.5 LTS and `gcc` version 11.4.0 (`Ubuntu 11.4.0-1ubuntu1~22.04`)
- You will have **eight** penalty-free submissions on Submitty, after which points will slowly be deducted, e.g., `-1` on submission #9, etc.
- You will have at least **three** days before the due date to submit your code to Submitty; if the auto-grading is not available three days before the due date, the due date will be 11:59PM three days after auto-grading becomes available

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` (or `drmemory` or other dynamic memory checkers) to check for errors with dynamic memory allocation and dynamic memory usage. As another helpful hint, close open file descriptors as soon as you are done using them.

Finally, always read (and re-read!) the `man` pages for library functions (section 3), system calls (section 2), 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 one or more input files. Your cache must be a dynamically allocated hash table of a given fixed size that handles collisions by simply replacing the existing word.

As part of this, you are required to provide a library function (in a `hw1.h` header file) that "tokenizes" an input string to dynamically identify each valid 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 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, Submitty will completely remove that line of code before running `gcc`.

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`.

As shown below, 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 in this way, the dereference `*` and address-of `&` operators negate one another.

Reusable tokenization function

You are required to provide a `tokenize()` library function in a `hw1.h` header file; nothing else should be in this header file. (Feel free to add other header files of your own.) Your `hw1.h` header file will be tested separately, which means it will be compiled in with other hidden test code.

The `tokenize()` function must have the following function prototype:

```
char ** tokenize( char * string );
```

This function "tokenizes" the given input `string` by identifying the start and end of each valid word. As described further on the next page, a word is any string of two or more alpha characters.

Specifically, this function constructs an array of `char *` pointers such that each element points to the next valid word in `string`. To know where this list of valid words ends, the last entry in this array must be `NULL`, similar to `argv[]` in the `command-line-args.c` example.

Your `tokenize()` function also must modify the given `string` by overwriting delimiter characters with `'\0'` to specify the end of each valid word.

As an example, consider the following input `string`:

```
"RPI) is a private research university in Troy, New York, United Sta"
```

There are 11 valid words, so an array of 12 `char *` pointers would be allocated and returned from this function, with the first pointer (at index 0) pointing to "`RPI`" in the original `string`, the second pointer (at index 1) pointing to "`is`" in `string`, etc., the third pointer (at index 2) pointing to "`private`" in `string`, etc., and the last pointer (at index 11) set to `NULL`.

Continuing the example, the original input `string` above is changed to:

```
"RPI\0 is\0a private\0research\0university\0in\0Troy\0 New\0York\0 United\0Sta"
```

Note that you must dynamically allocate memory for the `char *` array in your function. Use both `calloc()` and `realloc()` for this as you discover valid words. And it is up to the caller to free this dynamically allocated memory.

Finally, if an unrecoverable error occurs in this function, simply return `NULL`.

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. The cache size must be a positive integer.

Next, your program must open and read the regular file(s) specified by the remaining command-line arguments. Your program must parse and extract all words, if any, from each given file, in the given order the files are listed on the command line.

Here, a word is any string of two or more alpha characters; see below for how to "hash" a word. And if a collision occurs in your cache, simply replace the existing entry.

To read each input file, you **must** use `open()`, `read()`, and `close()`; you may also consider using `lseek()`. Any calls to library functions that make use of `FILE*` will be deactivated in Submitty; this includes `fopen()`, `fscanf()`, `fgets()`, etc.

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., `"lakers"`), 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.

Do **not** use `malloc()` or `memset()` 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 (as per `isalpha()`) and consisting of at least two characters. All other characters therefore serve as delimiters. And note that words are case insensitive, e.g., `Lion` is considered the same as `lion`; for output, always display words as lowercase.

Note that input files can be of any size and may have valid words at the beginning and/or end of the file, i.e., the file may begin and/or end in an alpha character.

To simplify your code, 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. **(v1.1)** Note that valid words should be converted to lowercase before hashing; words are to be stored as lowercase.

As an example, the valid word "`meme`" (**(v1.1)** regardless of case) consists of four ASCII characters, which sum to $109 + 101 + 109 + 101 = 420$. If the cache array size was 17, for example, then the array index for "`meme`" would be the remainder of 420/17 or 12.

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. **(v1.1)** Display the case of the original word. 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" ==> 13 (calloc)
Word "when" ==> 9 (calloc)
Word "Lion" ==> 9 (nop)
Word "was" ==> 8 (calloc)
Word "asleep" ==> 5 (calloc)
Word "little" ==> 8 (realloc)
Word "Mouse" ==> 9 (realloc)
Word "began" ==> 16 (calloc)
Word "running" ==> 4 (calloc)
Word "up" ==> 8 (realloc)
Word "and" ==> 1 (calloc)
...
```

Further, when you have finished processing all of the input file(s), show the contents of the cache by displaying a line of output for each non-empty entry in the cache. Use the following format (**(v1.1)** and note that words are stored in lowercase):

```
Cache:
[0] ==> "king"
[1] ==> "gnawed"
[2] ==> "plight"
[3] ==> "after"
[4] ==> "great"
[5] ==> "soon"
[6] ==> "to"
[7] ==> "tree"
[8] ==> "little"
[9] ==> "mouse"
[10] ==> "while"
[11] ==> "in"
[12] ==> "prove"
[13] ==> "beasts"
[14] ==> "not"
[15] ==> "the"
[16] ==> "friends"
```

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 perform final testing of your code), we will use Submitty.

To help make sure that your program executes properly, use the techniques below.

First, make use of the `DEBUG_MODE` preprocessor technique that helps avoid accidentally displaying extraneous output in Submitty. 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( "aaaaaaaaaaaaaggggggggghhhh square brackets!\n" );
#endif
```

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

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

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

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

You would not generally do this in practice since this can slow down your program, but to ensure you see as much output as possible in Submitty, this is a good technique to use.