CSCI 4210 — Operating Systems
Lecture Exercise 1 (document version 1.0)
Memory Management and File I/O in C

- This lecture exercise is due by 11:59PM EDT on Wednesday, May 31, 2023
- This lecture exercise consists of practice problems and problems to be handed in for a grade; graded problems are to be done individually, so **do not share your work on graded problems with anyone else**
- For all lecture exercise problems, take the time to work through the corresponding course content to practice, learn, and master the material; while the problems posed here are usually not exceedingly difficult, they are important to understand before attempting to solve the more extensive assignments in this course
- 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`)

Practice problems

Work through the practice problems below, but do not submit solutions to these problems. Feel free to post questions, comments, and answers in our Discussion Forum.

1. Run the `static-allocation-buggy.c` example on your own Linux platform(s), not necessarily Ubuntu. With each of the initial `char` arrays set to a size of 5 bytes, what is the **exact** terminal output? Does this differ from what we saw in class on Ubuntu?

   Add code using the `"%p"` conversion specifier to display the memory address of each array in this example. Remember that these are statically allocated arrays.

   Next, change the size of these two `char` arrays to be 6 bytes and re-run your code. Try reducing to only 3 bytes. Also try other values.

   Is it possible, by only changing the hard-coded size of these first two arrays, to cause a segmentation fault? If so, how? If not, why not?

2. Run the `dynamic-allocation.c` example code on your own Linux platform(s), again not necessarily Ubuntu. When run with `valgrind`, why are there three dynamic memory allocations when the code only has two `malloc()` calls? How many bytes are dynamically allocated in this third "hidden" call to `malloc()`?

   When you run the `dynamic-allocation.c` example, do you see the same output on your Linux instance? If not, why not?

   Why does the output appear as shown below, in particular starting on the fifth line of output?

```
sizeof path is 8
path is "/cs/goldsd/u23/" (strlen is 15)
path2 is "ABCDEFGHIJKLMNOP" (strlen is 16)
path is "/cs/goldsd/u23/" (strlen is 15)
path2 is "/cs/goldsd/u23/blah/BLAH/blAh/blpath2 is "cs/goldsd/u23/" (strlen is 15)
)
" (strlen is 75)
```

   Add code using the `"%p"` conversion specifier to display the memory address of each pointer variable in this example. Also use `"%p"` to display the memory addresses pointed to on the runtime heap.

   Modify the given code by correcting it such that it allocates the minimum number of bytes required for `path` and `path2`. Test this via `valgrind` (or `drmemory`) to be sure there are no invalid reads or writes.

3. In the code below, how many bytes are dynamically allocated for each call to `malloc()` and `calloc()`?

   For each expression in which we use pointer arithmetic, how many bytes are added to the original pointer?

   What is the exact output of the given code? Can we definitively predict the output or are some values unpredictable?

   Finally, add the appropriate calls to `free()` to ensure no memory leaks.

```
char * name = malloc( 16 );
int * x = calloc( 2, sizeof( int ) );
int * numbers = calloc( 32, sizeof( int ) );
double * values = calloc( 32, sizeof( double ) );

sprintf( name, "ABCD-%04d-EFGH", *x );
printf( "%s\n", name + 3 );
printf( "%d", *(numbers + 5) );
printf( "%lf\n", *(values + 5) );
```

Graded problems

Complete the problems below and submit via Submitty for a grade. Please do not post any answers to these questions. All work on these problems is to be your own.

1. For this problem, place all of your code in a `reverse.h` header file. Only submit this header file on Submitty.

   This file will be included by hidden code on Submitty and compiled as follows:

   ```
   gcc -Wall -Werror lecex1-q1.c
   ```

   Review the `reverse()` function shown below to first understand what it does.

   ```
   char * reverse( char * s )
   {
     char buffer[32];
     int i, len = strlen( s );
     for ( i = 0 ; i < len ; i++ ) buffer[i] = s[len-i-1];
     for ( i = 0 ; i <= len ; i++ ) s[i] = buffer[i];
     return s;
   }
   ```

   Rewrite the `reverse()` function by using dynamic memory allocation for `buffer`.

   More specifically, do away with the 32-byte buffer and allocate **exactly** the number of bytes that you need to store the string. Be sure that you check your solution via `valgrind` (or `drmemory`).

   Also, replace all square brackets with pointer notation, i.e., only use pointer arithmetic in your implementation. **Code containing square brackets, including comments, will be automatically deleted on Submitty!**

   Were you able to write a solution that compiles and runs without error on the first try?

2. Write a program called `chunk.c` that accepts two command-line arguments `n` and `filename`. Use `open()`, `read()`, `lseek()`, and `close()` to extract `n`-byte chunks from `filename`, skipping every other chunk by using `lseek()`.

   Refer to the `man` pages to better understand these functions.

   Display every other chunk to `stdout`, delimiting them with a pipe `'|'` character. Always append one newline `'\n'` character at the end of your program execution.

   As an example, given an input file consisting of the English alphabet in uppercase, plus a newline character, your output should produce the results shown below.

```
bash$ cat infile.txt
ABCDEFGHIJKLMNOPQRSTUVWXYZ
bash$ wc -c infile.txt
27 infile.txt
bash$ ./a.out 7 infile.txt
ABCDEFG|OPQRSTU
bash$ ./a.out 4 infile.txt
ABCD|IJKL|QRST|YZ

bash$
```

   In the example output above, note that the last chunk printed is `"YZ\n"`. Be sure to read the `man` pages for the `cat` and `wc` programs shown in the example above.

   Just as you did with Question 1, replace all square brackets with pointer notation (i.e., use pointer arithmetic). **Code containing square brackets, including comments, will be automatically deleted on Submitty!**

   Be sure there are no memory leaks or warnings.

   As a hint, use `atoi()` to convert a string into an integer. Further, use the return value of `read()` to determine when to stop.

   Once again, were you able to write a solution that compiles and runs without error on the first try?

What to submit

Please submit two C source files called `reverse.h` and `chunk.c`. They will be automatically compiled and tested against various test cases, some of which will be hidden.