

CSCI 4210 — Operating Systems  
Lecture Exercise 1 (document version 1.1)  
Dynamic Memory Allocation, System Calls, and Files (SOLUTIONS)

- This lecture exercise is due by 11:59PM ET on ~~Wednesday~~ Thursday, February 3 4, 2021
- 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 video lecture(s) 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 homeworks in this course
- As with the homeworks, 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 Submittity, which uses Ubuntu v18.04.5 LTS and `gcc` version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)

## 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.c` example from the January 25 video lecture on your own Linux platform (not necessarily Ubuntu). Do you see the same output? If not, why not?

How would the addition of reading in input from the user potentially mimic the incorrect buffer behavior shown in this example?

**SOLUTION:**

Reading in input from the user could cause buffer overflow, in particular if we use `scanf()` as follows:

```
scanf( "%s", buffer );
```

Look for other ways (by reading the `man` pages of other library functions and system calls) to read input from the user. Pay close attention to the potential for buffer overflow.

2. Run the `dynamic-allocation.c` example from the January 25 video lecture on your own Linux platform (again, not necessarily Ubuntu). Do you see the same output? And if not, why not?

Also, why does the output appear as shown in the video (and pasted below), in particular starting on the third line of output?

```
path is /csci/goldsd/s21/  
path2 is AAAAAAAAAAAAAAA  
path is /csci/goldsd/s21/blah/blah/blah/path is  AAAAAAAAAAAAAAA  
/  
  
path2 is AAAAAAAAAAAAAAA
```

### **SOLUTION:**

The output appears as shown above due primarily to a buffer overflow. When we modify `path` to be the longer string, we go beyond the allocated 20 bytes, corrupting the runtime heap.

The strange output occurs, though, because some library functions and system calls might also dynamically allocate memory. In this case, `printf()` dynamically allocates memory (likely to build the string to be output). Try commenting out the first `printf()` statement and seeing what happens.

And in general, try commenting out and/or changing the order of some of the lines of code, then observing the results. Also compile with the `-g` option of `gcc`, then run `valgrind` to more closely trace the memory errors.

## Graded problems

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

1. Study the `reverse()` function shown below to understand how it works.

```
char * reverse( char * s )
{
    char buffer[1024];
    int i, len = strlen( s );
    for ( i = 0 ; i < len ; i++ ) buffer[i] = s[len-i-1];
    buffer[i] = '\0';
    strcpy( s, buffer );
    return s;
}
```

Rewrite the `reverse()` function by using dynamic memory allocation for `buffer`. Use `malloc()` for your dynamic memory allocation, and be sure there are no memory leaks.

Submit this code in a file called `reverse.h` (and only include the function itself).

### **SOLUTION:**

The intended solution is shown below (with new lines of code indicated):

```
char * reverse( char * s )
{
    /*    char buffer[1024];    <== REMOVE THIS LINE */
    int i, len = strlen( s );
==> char * buffer = malloc( len + 1 );
    for ( i = 0 ; i < len ; i++ ) buffer[i] = s[len-i-1];
    buffer[i] = '\0';
    strcpy( s, buffer );
==> free( buffer );
    return s;
}
```

2. Write a program called `extract.c` that uses `open()`, `read()`, `lseek()`, and `close()` to extract every seventh byte from the input file specified as a command-line argument. Display these individual bytes to `stdout`. And only append one newline `'\n'` character to `stdout` 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 appear as shown below (i.e., the output of `./a.out`).

```
bash$ cat infile.txt
ABCDEFGHJKLMNOPQRSTUVWXYZ
bash$ wc -c infile.txt
27 infile.txt
bash$ ./a.out infile.txt
GNU
bash$
```

As a hint, use the return value of `read()` to determine when to stop.

**SOLUTION:**

See the `extract.c` and corresponding lecture video for February 8 for a solution and related discussion.