1. The purpose of this assignment is to use UNIX system calls to do some very basic process operations, and to retrieve arguments from the command line. You must write a program that accepts two command line arguments (a sleep time and an iteration count). This program will contain a loop which sleeps for the specified amount of time, then wake up and print its name, process ID (PID) and iteration count to **stderr**. The loop should execute the number of times specified in the iteration count. Then the program should print a message and terminate.

   For example, if the name of your program is **c1.c** , the following command should execute the program for 10 iterations, sleeping 4 seconds between iterations.

```
./c1 4 10

Executing c1, process id 28435, iteration number 1
                          ...
Executing c1, process id 28435, iteration number 10
c1 is now exiting
```

2. One of the operations the shell performs is to parse a string into tokens. When you run your program with ./a.out argc will have the value of 1, and argv[0] will have the value of ./a.out. Understanding how to tokenize a string is an important concept. For this question you will emulate the parsing the shell does. You will need to write several functions, and a comprehensive tester

   Write a function that will parse a string into tokens (or words), similar to what the shell is required to do. The function is named makeargs. The prototype is given below.

   *int makeargs(char * s, char *** args);* or *int makeargs(char s[], char **args[]);*

   This function should accept a (c-type) string and an array of pointers to char (i.e., the same type as argv in a C program), and should return the pointer (defined to point to an array pointing to the separate tokens on the command line) argv and the number of tokens that are in the array argv. If a problem occurred during operation of the function, then return -1.

   For example, given the following C code

```
int main()
{
   char **argv, s[] = "ls -l file";

   int argc;

   argc = makeargs(s, &argv);

   printargs(argc, argv);

}// end main
```

   The results of *makeargs* would be:

   argc would be 3.
   argv[0] would be 'ls'
   argv[1] would be '-l'
   argv[2] would be file

   You must not waste memory, and any memory you allocate you must clean up.

   I have provided as a starting point c2.c

   NOTE You will need to change the main function to allow the user to enter strings. The strings will be entered on the command line separated by a single space. You can presume the happy part of Stuland. HINT: You may need strtok and other string commands.

   DO NOT use **realloc,** use only **free** and **malloc**/**calloc**

   For example you might prompt the user with something like:

   Please enter a bunch of strings, each string will be separated by a space

   The user would enter: how now brown cow

And your program would report 4 strings and then print each string.

This will continue until the user chooses to stop. You can do this with a menu, or a letter or whatever as long as the user is prompted to quit/continue.

You must use strtok_r from string.h

Before changing your main make sure everything is working as expected. Your output capture should have multiple outputs showing you truly tested your program. NOTE: the grader will be running **valgrind** on your program so make sure you do clean up memory.

3. In class we discussed the basics of fork and exec. To help you understand them, let's practice. I have provided a C file named c3.c. Your tasks are:
   a. Take the makeargs code from prob 2 and place it in the appropriate place Your makeargs must have one more row that is NULL.
   b. Take the cleanup code from prob 2 and place it in the appropriate position
   c. Add the appropriate #includes to c3.c – You will need to examine the man pages for fork, exec and wait
   d. Write the forkIt function. This function will:
      i. fork() and store the result
      ii. check the result of fork() if in the parent - issue a waitpid command as illustrated in the man page
      iii. if the result of the fork() indicates the child - issue the appropriate exec command
      iv. Check the results of the exec command – if invalid then the invalid exec command will be appropriately dealt with so execution can continue – simply put, I should not have to type exit more than once to get the program to end!