Background

To complete these exercises, you will need to learn about the following functions and system calls.

- `read()` — the system call
- `execvp()` — invokes `execve`; uses an argument vector and searches `PATH`
- `strtok()` — isolates tokens in a string

Exercise 1. In class, we saw how to use the `strtok()` function — see Listing 1. Although this is one of the more unusual string functions provided by the string library, it is quite handy.

Get a copy of this program. Study it carefully, then compile and run it. Change the string constant (*Four score...*) in various ways and re-run the program. Be sure to test for extreme cases, including a line with no words and a line with just one word.

You will need to write similar code later on, so your goal should be to convince yourself that this program correctly isolates all the words in a single line of text.

Exercise 2. Make a copy of this program, naming it `strtok-demo2.c`. Remove the `printf()` statement from within the loop. Introduce a new data structure, capable of storing a collection of strings:

```
char *words[MAX_SIZE];
```

Modify your program so it stores the words being isolated by `strtok()`, placing each one in your array. To convince yourself that you have done this correctly, output the words in reverse order.

Exercise 3. One of the popular shells on Unix systems is the "Bourne-again shell," known more simply as `bash`. By knowing just a few system calls, we can implement our own shell — though it will have far fewer capabilities. For this exercise, you will implement `ish`, an "incredibly simple shell."

Study the code in Listing 2. This program will compile and run, though it doesn't do very much. Consult the Unix manual page for the `read()` system call, then observe how it is being used in this program. Complete this program as follows:

1. Get a copy of the source code, then compile and run it. Observe that control-D will terminate this program. Try a few sample inputs.
2. Using what you learned in earlier exercises, add code to the `readAndParseArgs()` function in order to create and save an argument vector. Make sure this vector is properly terminated with a `NULL` entry.
3. Complete the `displayArgs()` function, then use it to ensure that your argument vector is being created correctly.
4. Extend `main()` so that it acts on the command (and, optionally, command line arguments). Use `fork()`, `execvp()`, and `wait()`.

What to Submit

Put copies of your completed C programs into a folder, then drag this folder onto the EIU submit icon. Be sure that your name appears in each program as a comment somewhere near the top of the file. Your programs should be generously commented.

Source Code

```
/*
  Demonstration of the strtok() function

  MAT 4970
  Bill Slough
*/

#include <stdio.h>
#include <string.h>

/* What characters are used to separate words? */
#define DELIMITERS " "

int main() {
    /* A simple string for illustration */
    char line[] = "   Four score and seven years ago our fathers brought forth";

    /* A pointer to be used by strtok() */
    char *ptr;

    printf("Before processing: \"%s\"\n", line);

    /* Find the first word in the line */
    ptr = strtok(line, DELIMITERS);

    while (ptr != NULL) {
        /* process the current word */
        printf("\"%s\"\n", ptr);

        /* get the next word in the line */
        ptr = strtok(NULL, DELIMITERS);  /* NB: line is NOT the first argument! */
    }

    /* Observe that strtok() modifies the string we have been scanning */
    printf("After processing: \"%s\"\n", line);

    return 0;
}
```

Listing 1: A program to find the "words" in a string.

```c
/*
  ISH: Incredibly Simple Shell

  Author:
  MAT 4970
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define MAX_LINE_LENGTH 80                /* upper bounds on line */
#define MAX_ARGUMENTS MAX_LINE_LENGTH/2   /* and number of arguments */

typedef int bool;
#define FALSE 0
#define TRUE 1

void readAndParseArgs(char inputBuffer[], char *args[]);
void displayArgs(char *args[]);

int main(void) {
    char inputBuffer[MAX_LINE_LENGTH + 1];    /* buffer to hold the command entered */
    char *args[MAX_ARGUMENTS + 1];            /* array of arguments */

    while (TRUE) {
        /* Provide a prompt for the command line */
        printf("ISH> ");
        fflush(stdout);    /* force output to appear */

        /* Get a command line */
        readAndParseArgs(inputBuffer, args);

        /* Show the individual arguments */
        displayArgs(args);

        /* Create a child process, execute the specified command,
           and wait for completion */
    }
}

void readAndParseArgs(char inputBuffer[], char *args[]) {
    int length;    /* number of characters read from the input */

    /* get a line of input from the standard input file */
    length = read(STDIN_FILENO, inputBuffer, MAX_LINE_LENGTH + 1);
    if (length == 0)
        exit(0);             /* EOF encountered ... quit the shell */

    /* Transform the inputBuffer to a C-style string */
    inputBuffer[length - 1] = '\0';

    printf("Line entered = \"%s\"\n", inputBuffer);

    /* Isolate the arguments found on the input line and
       store them in the argument vector */
}

void displayArgs(char *args[]) {
    /* Output the arguments in a given argument vector */
}
```

Listing 2: An outline of a bare-bones shell.