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

#define MAX_LINE 80 /* 80 chars per line, per command, should be enough. */

/**
 * Performs a recursive directory listing
 */
void do_rec(char *dir_name, int times)
{
    DIR *dir_ptr;
    char *new_name;
    struct dirent *dirent_ptr;
    int i;

    for (i = 1; i < times; i++)
        printf("   ");
    printf("Now listing directory -- %s:
", dir_name);
    if ((dir_ptr = opendir(dir_name)) != NULL){
        while ((dirent_ptr = readdir(dir_ptr)) != NULL){
            for (i = 0; i < times; i++)
                printf("   ");
            printf("%s
", dirent_ptr->d_name);
        }
        closedir(dir_ptr);
    }

    if ((dir_ptr = opendir(dir_name)) != NULL){
        while ((dirent_ptr = readdir(dir_ptr)) != NULL){
            if (dirent_ptr->d_type == DT_DIR){
                if (strcmp(dirent_ptr->d_name, ".") == 0 ||
                    strcmp(dirent_ptr->d_name, "..") == 0)
                    continue;
                new_name = (char *)malloc(strlen(dirent_ptr->d_name) + strlen(dir_name) + 2);
                strcpy(new_name, dir_name);
                if (strcmp(dir_name, "/") != 0)
                    strcat(new_name, "/");
                strcat(new_name, dirent_ptr->d_name);
                do_rec(new_name, times + 1);
            }
        }
        closedir(dir_ptr);
    }
} /* end of do_rec */

/* The setup function below will not return any value, but it will just: read
in the next command line; separate it into distinct arguments (using blanks as
delimiters), and set the args array entries to point to the beginning of what
will become null-terminated, C-style strings. */

int setup(char inputBuffer[], char *args[], int *background)
{
    int length, /* # of characters in the command line */
    i, /* loop index for accessing inputBuffer array */
    start, /* index where beginning of next command parameter is */
ct;  /* index of where to place the next parameter into args[] */

c = 0;

/* read what the user enters on the command line */
length = read(STDIN_FILENO,inputBuffer,MAX_LINE);

/* 0 is the system predefined file descriptor for stdin (standard input),
which is the user’s screen in this case. inputBuffer by itself is the
same as &inputBuffer[0], i.e. the starting address of where to store
the command that is read, and length holds the number of characters
read in. inputBuffer is not a null terminated C-string. */

start = -1;
if (length == 0)
    exit(0);  /* ^d was entered, end of user command stream */

/* the signal interrupted the read system call */
/* if the process is in the read() system call, read returns -1
However, if this occurs, errno is set to EINTR. We can check this value
and disregard the -1 value */
if ( (length < 0) && (errno != EINTR) ) {
    perror("error reading the command");
    exit(-1);  /* terminate with error code of -1 */
}

for (i=0;i<length;i++) {  /* examine every character in the inputBuffer */
    switch (inputBuffer[i]){  
    case ' ':  
        case 't':  /* argument separators */
            if(start != -1){
                args[ct] = &inputBuffer[start];  /* set up pointer */
                ct++;
            }
            inputBuffer[i] = '\0'; /* add a null char; make a C string */
            start = -1;
            break;

        case 'n':  /* should be the final char examined */
            if (start != -1){
                args[ct] = &inputBuffer[start];
                ct++;
            }
            inputBuffer[i] = '\0';
            args[ct] = NULL;  /* no more arguments to this command */
            break;

        default:  /* some other character */
            if (start == -1)
                start = i;
            if (inputBuffer[i] == '&'){  /* background = 1 */
                inputBuffer[i-1] = '\0';
            }
        }
    }
}

/* perform a recursive directory listing */
if (strncmp("RLS",inputBuffer,3) == 0) {
    do_rec(args[1], 1);
return 1;
}

return 0;
} /* end of setup routine */

int main(void)
{
char inputBuffer[MAX_LINE]; /* buffer to hold the command entered */
int background; /* equals 1 if a command is followed by '& */
char *args[MAX_LINE/2 + 1]; /* command line (of 80) has max of 40 arguments */
int child, /* process id of the child process */
status; /* result from execvp system call*/
int isRLS; /* indicates if RLS command is being run */

while (1) { /* Program terminates normally inside setup */
    background = 0;
    printf("COMMAND-\n");
    isRLS = setup(inputBuffer,args,&background); /* get next command */

    if (!isRLS) {
        child = fork(); /* creates a duplicate process! */
        switch (child) {

            case -1:
                perror("could not fork the process");
                /* perror is a library routine that displays a system
                 * error message, according to the value of the system
                 * variable "errno" which will be set during a function
                 * (like fork) that was unable to successfully
                 * complete its task. */
            
                break;

            case 0: /* here is the child process */
                status = execvp(args[0],args);
                if (status != 0){
                    perror("error in execvp");
                    exit(-2); /* terminate this process with error code -2 */
                }
                break;

            default: /* this is the parent */
                if(background == 1) /* handle parent,wait for child */
                    while (child != wait(NULL)) ;
        }
    }
}
}