Library functions

This assignment introduces you to writing functions that utilize the C calling convention (cdecl), and packaging those functions together to create a library of functions that can be used by somebody else's code.

To test your code, you may use the two files at the end of this document **start.asm** and **main.c**. start.asm is a simple _start entry point routine that marshals parameters and calls main. Once main returns, it simply calls exit using main's return value as the status parameter for **exit**. The example main.c program doesn't do too much, some basic input and output, and a couple of string operations. It makes no attempt to thoroughly exercise all of the code that you will write. **It is your responsibility to test your code until you are satisfied that it is ready to turn in**. This means you should modify the existing main.c or write an entirely new one and include code to test your functions.

Your task is to implement all of the functions described below utilizing only assembly language. Your code may make system calls using 'syscall', but it may not call any functions that you have not written yourself. Your code will be evaluated based on its interoperability with the instructor's main.c. Once complete, link your file together with start.o and main.o to create a 64-bit Linux executable. Assuming your file is properly named assign4.asm, you should be able to build with the following commands:

```
nasm -f elf64 start.asm nasm -f elf64 assign4.asm

gcc -o assign4 –m64 main.c assign4.o start.o -nostdlib \
-nodefaultlibs -fno-builtin -nostartfiles
```

The options instruct gcc not to link to any code other than what you name on the command line which means you need to provide your own _start (provided to you), your own main (provide to you or written yourself) and library routines (provided by you).

You should then be able to run your program from the command line by typing:

```
./assign4
```

Here are the prototypes and behaviors of the functions that you need to implement in assembly language. Don't forget to include a global statement for each function in assign4.asm:

```
int l_strlen(char *str);
```
Return the length of the null terminated string, str. The null character should not be counted.

```
int l_strcmp(char *str1, char *str2);
```
Return 0 if str1 and str2 are equal, return 1 if they are not. Note that this is not quite the same definition as the C standard library function strcmp.

```
int l_gets(int fd, char *buf, int len);
```
Read at most len-1 bytes from file descriptor fd, placing them into buffer buf. If len == 0, then read nothing and return 0. Terminate early if a new line character ('\n', 0x0A) character is read. If a new line character is encountered, it should be stored into the output buffer and counted in the total number of bytes read.

Return the total number of bytes read (which may be zero if end of file is reached or an error occurs). This function must place a null termination character after the last character read. The null termination character is not counted as part of the return count.

`void l_puts(const char *buf);`

Write the contents of the null terminated string buf to stdout (file descriptor 1). The null byte must not be written. If the length of the string is zero, then no bytes are to be written.

`int l_write(int fd, char *buf, int len, int *err);`

Write len bytes from buffer buf to file descriptor fd. **Return the number of bytes actually written or -1 if an error occurs**. If an error occurs, the absolute value of the kernel-supplied error code must be returned in *err. If no error occurs, then *err should be set to zero.

`int l_open(const char *name, int flags, int mode, int *err);`

Open the named file with the supplied flags and mode. **Returns the integer file descriptor of the newly opened file or -1 if the file can't be opened**. If an error occurs, the absolute value of the kernel supplied error code must be returned in *err. If no error occurs, then *err should be set to zero.

`int l_close(int fd, int *err);`

Close the indicated file. **Returns 0 on success or -1 on failure**. If an error occurs, the absolute value of the kernel-supplied error code must be returned in *err. If no error occurs, then *err should be set to zero.

`void l_exit(int rc);`

Terminate the calling program with the exit code rc.

IMPORTANT NOTE: For correctly implementing the 8 functions listed above, you will receive a maximum grade of * 79 ***. Correct implementation of each of the 3 functions listed below will earn a maximum of 7 additional points per correctly implemented function.**

`unsigned int l_atoi(char *value);`

Perform an ASCII to decimal conversion of the decimal number contained in value. Valid characters in value include '0'..'9'. ANY other character is considered invalid. Your function must return the 4-byte, little endian representation of all consecutive decimal digits from the beginning of the string. You must stop the conversion when any character other than '0'..'9' is reached (including the null termination byte). If the very first character in the string is invalid you must return 0. Some examples:

```
"123" returns 123
"456abc" returns 456
"abc789" returns 0
"" returns 0
```

`char *l_itoa(unsigned int value, char *buffer);`

Perform the decimal to ASCII conversion of value, placing the null terminated ASCII content into buffer. You may assume that buffer is of sufficient size to hold all required characters including the null terminator (minimum of 10 bytes). This function should return a pointer to the generated buffer (ie return buffer). Some examples:

```
   123 generates "123"
   l_puts(l_itoa(456, buffer)); //should print 456 to stdout

unsigned int l_rand(unsigned int n);
```

Generate a random number in the range 0..n-1 inclusive using the following process:

1. Open "/dev/urandom" (perhaps using l_open)
2. Read 4 bytes from the open file (these will be a random number in the range 0..0xffffffff). Call this number: r.
3. Close the open file.
4. Return r % n.

Since all of these functions will be called from a C program, they **must** adhere to the cdecl calling convention. Keep in mind that according to the x86_64 Application Binary Interface (ABI), functions are allowed to corrupt the contents of rax, rdi, rsi, rcx, rdx, r8, r9, r10, and r11, while they must preserve the state of all other registers. Your functions **must** all adhere to this convention.

Your program must be thoroughly commented. Endline comments must be justified and must not wrap; start a new line instead. Use standalone comments to describe complex logic.

The main.c file below is ***only an example*** to assist you in creating a working executable. You may want to add additional code to ensure that all functions that you have implemented are thoroughly tested.

Deliverables:

Your assembly language source file which must be named **exactly assign4.asm**. Your source file MUST contain a header comment that includes your name, the course number, and the assignment number. Submit this file to the assignment submission page in Sakai. **Do not turn in start.asm or main.c.**

```
;start.asm

bits 64

extern main
global _start

section .text

_start:
   lea   rsi, [rsp + 8]
   mov   rdi, [rsp]
   lea   rdx, [rsi + rdi * 8 + 8]
   call  main
   mov   rdi, rax
   mov   eax, 60
   syscall
```

```
//main.c
int l_strlen(const char *);
int l_strcmp(const char *str1, const char *str2);
int l_gets(int fd, char *buf, int len);
void l_puts(const char *buf);
int l_write(int fd, const char *buf, int len, int *err);
int l_open(const char *name, int flags, int mode, int *err);
int l_close(int fd, int *err);
unsigned int l_atoi(char *value);
char *l_itoa(unsigned int value, char *buffer);
unsigned int l_rand(unsigned int n);
int l_exit(int rc);

#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR   2

char prompt[] = "Enter a string: ";
char msg1[] = "Echoing file: ";
char msg2[] = "Failed to open input file\n";
char equal[] = "The strings are the same\n";
char diff[] = "The strings are different\n";
char value[] = "The second string contains the integer: ";
char rnd[] = "Here is a random number between 0 and 4000 for you: ";
```

```
int main(int argc, char **argv, char **envp) {
   int i;
   char newline = '\n';
   int len1, len2;
   char str1[80];
   char str2[80];
   int fd;
   int err;

   for (i = 0; i < argc; i++) {
      l_puts(argv[i]);
      l_write(1, &newline, 1, &err);
   }
   l_write(1, &newline, 1, &err);

   if (argc > 1) {
      l_puts(msg1);
      l_puts(argv[1]);
      l_write(1, &newline, 1, &err);
      fd = l_open(argv[1], O_RDONLY, 0, &err);
      if (fd == -1) {
         l_puts(msg2);
      }
      else {
         int len;
         while ((len = l_gets(fd, str1, 79)) > 0) {
            l_write(1, str1, len, &err);
         }
         l_close(fd, &err);
      }
   }

   l_write(1, &newline, 1, &err);

   while (1) {
      l_write(1, prompt, l_strlen(prompt), &err);
      len1 = l_gets(0, str1, 79);
      str1[len1] = 0;

      if (l_strcmp(str1, "quit\n") == 0) {
         break;
```

```
      }

      l_puts(prompt);
      len2 = l_gets(0, str2, 79);
      str2[len2] = 0;

      if (l_strcmp(str1, str2) == 0) {
         l_puts(equal);
      }
      else {
         l_puts(diff);
      }

      l_puts(value);
      l_puts(l_itoa(l_atoi(str2), str1));
      l_write(1, &newline, 1, &err);

      l_puts(rnd);
      l_puts(l_itoa(l_rand(4000), str1));
      l_write(1, &newline, 1, &err);

   }
   l_exit(0);
   return 0;
}
```