operating systems lab - week 6: exercise

Nicolo Colombo

November 3, 2020

This lab is about creating linked lists of words. Structures and dynamic memory allocation are the two basic building blocks that you need to complete this exercise.

Please try to complete all sections of this lab sheet before attempting this week's Moodle quiz

lab quiz - week 6

as some of the quiz questions may require you to run, comment or modify the programs you are asked to write.

Set up

Depending on your OS, use the following instructions to connect to `linux.cim.rhuk.ac.uk`:

Unix Open the terminal and run

```
ssh yyyyxxx@linux.cim.rhul.ac.uk
```

where `yyyyxxx` is your college username, and enter your password to access the teaching server.

Windows Launch the Windows SSH client `puTTY` [1], enter the following

```
linux.cim.rhul.ac.uk
```

in the empty field *Host Name (or IP address)* and click on *Open*. The client opens a new window where you are required to enter your college user name `yyyyxxx` and password.

Once logged in, you should be able to see the content of and navigate in your home directory using the standard UNIX commands, e.g. `ls`, `cd`, `cp`. Go to the `CS2850labs` directory[2] and run the command

```
$mkdir week6
```

to create a new sub-directory called `week6`. We suggest you save and compile all programs you write for this exercise in this directory.

Use a command-line text editor, e.g. emacs, nano, or vim,

to open, edit and save your programs. The advantage of command-line editors is that they can be used in a non-graphical SSH session. [3] You can create a new C file or open an existing C file, `file_name.c`, by running the command

```
$editorName file_name.c
```

where, e.g. `editorName` is `vim`

We suggest you save separate files for all single parts of this exercise and follow the name suggestions given in each section. [4]

Compile your C code by running

```
$clang -o file_name file_name.c
```

[1] `puTTY` should be installed on all department's machines. If you work on your own Windows machine you can download it at download puTTY and install it as explained.

[2] The parent directory you have created for the first week's lab

[3] If you do not want to open and close the editor every time you modify and save your code you can open a new SSH session and use two shell windows simultaneously.

[4] This is mainly because some of the Moodle quiz questions may refer to single pieces of code through the suggested file names.

and run the corresponding binary files `file_name` through

`$./file_name`

For debugging, we suggest you use the free debugging tools of valgrind, which is already installed on the teaching server `linux.cim.rhul.ac.uk`. To check your code, you just need to run the command

`$valgrind ./file_name`

and have a look at the messages printed on the terminal.

1 malloc

In this section, you will write a program, `mallocWords.c`, that stores a series of input words (entered by the user) in a array of strings dynamically allocated in the heap using `malloc`. The size of the array will be obtained at runtime by counting the number of words in the input. A drawback of this approach is that you need to assume the input's maximum size (in number of character).[5] We suggest you fix the input's maximum size at the beginning of your program by including

`#define MAXCHARS 500`

The idea is to use `malloc` to allocate a string array that contains all the input words after you have counted them.

To save the input into a string, `s`, and compute its length, use

`int readLine(char *s, int maxChars)`

and

`int stringLength(char *s)`

that you have written to complete Section 2 of Week 3's exercise.

In Week 3, you assumed that the user's input has

- a fixed maximum number of characters, `MAXCHARS` and
- a fixe maximum number of words, `MAXWORDS`,

stored the whole input into a string, `s`, split `s` into single words, and loaded the word *starting addresses* into the entries of a pointer array, by calling

`int getWords(int nWords, char **t, char *s)`

where you set the `nWords` parameter to `MAXWORDS`.

Here, you will use dynamic memory allocation to drop the second constraint and allow the number of words to be arbitrary[6]. For allocating the right amount of memory in the *heap*, you will need to know the number of words in the user's input. Declare an integer variable,

`int nWord;`

in `main`, and use the following function to set its value

```
int countWords(int *nWords, char *s){
        int lim = stringLength(s);
        *nWords = 0;
        int sl = 0;
        int j = 0;
        while (j < lim){
                sl = 0;
                while ((*s!= ' ') && (*s != '\n') && (j < lim)){
                        s++;
                        j++;
                        sl++;
                }
                s++;
                j++;
                if (sl > 0)(*nWords)++;
```

[5]You will see how to remove this constraint in Sections 2 and 3

[6]Provided their total length plus the number of empty spaces is less than `MAXCHARS`

```
        }
        return j - 1;
}
```

The return value of `countWords` is the length of the input string Compare it with the return value of `readLine` to check that your functions behave correctly. Before including `countWords` into your program, try to understand how it works by calling it from `main` and printing the updated value of `nWord`. For example, what is the role of the last `if` statement?

Once you have set the number of words you can proceed as in Week 3 by declaring a pointer array, `t`, for storing the start of each word in `s`. This time the size of the pointer array is fixed at runtime and you cannot be declared as you did in Week 3. Declare `t` as a pointer to a pointer to `char`, i.e. declare `t` as

```
char **t;
```

and make it point to a heap memory block of the right size by letting

```
t = malloc(nWords * sizeof(*t);
```

Now you can use[7]

```
int getWords(int nWords, char **t, char *s)
```

to split `s` into words and load their starting points into `t`.

Finally, to print the content of the string array, use

```
void printWords(char *s, char **t, int size){
        for(int k = 0; k < size; k++){
              printf("%d-th word: %s (%d)\n", k + 1, *(t + k), stringLength(*(t+k)));
   }
}
```

Template As `main`, you can use the following template

```
int main(){
        char s[...];
        int count = ...;
        int sLength = readLine(...);
        int sLength2 = countWords(...);
        if (...) return -1;
        char **t = malloc(...);
        int size = getWords(...);
        printWords(...);
        free(...);
}
```

Example A run of the program with user input

```
one Two three Four five Six seven Eight nine Ten
```

should produce the following output:

```
1-th word: one (3)
2-th word: Two (3)
3-th word: three (5)
4-th word: Four (4)
5-th word: five (4)
6-th word: Six (3)
7-th word: seven (5)
8-th word: Eight (5)
9-th word: nine (4)
10-th word: Ten (3)
```

[7]The same function that you have written in Week 3.

If you execute your program with Valgrind (and the same `stdin` input as above), you should not see any error or leaking messages, i.e. Valgrind should print something similar to

```
==3231246==
==3231246== HEAP SUMMARY:
==3231246==     in use at exit: 0 bytes in 0 blocks
==3231246==   total heap usage: 3 allocs, 3 frees, 2,128 bytes allocated
==3231246==
==3231246== All heap blocks were freed -- no leaks are possible
==3231246==
==3231246== For lists of detected and suppressed errors, rerun with: -s
==3231246== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Note This week's quiz will ask you to run the program you wrote in this section. Before proceeding, be sure that your code reproduces exactly the output shown above when you use

`one Two three Four five Six seven Eight nine Ten`

as `stdin` input.

2 realloc

In this section, you will write a program that stores the input words in a dynamically allocated array of strings that grows while parsing the user's input.[8] The advantage of this version is that only the maximum length of the words should be fixed but no maximum length of the whole user's input. More concretely, you will need to process the input in a slightly different way and call `realloc` to adjust the size of the string array each time a new word arrives. In particular, you will need to write a new function that reads the input word by word.

You can assume that the maximum length of a word is fixed, i.e. you can write

`#define MAXLENGTH 100`

at the beginning of your file [9] and declare a buffer variable, `buf`, as

`char buf[MAXLENGTH];`

to store the current word. To read a single word from the input you can use

```
int getWord(char *buf, int *end, int maxLength){
        int j = 0;
        char c;
        while (((c = getchar()) != ' ') && (c != '\n') && (j < maxLength)){
                buf[j++]=c;
                }
        if (c == '\n') *end = 1;
        return j;
}
```

where the `end` argument is used to stop reading and loading word when you reach the end of the input, i.e. a new line character.

To re-allocate memory in the heap using

`void* realloc(void* p, int size)`

you need the `p` argument[10] to point to some memory block that is also in the heap. In other words, you can *not* declare the empty string array, `t`, that will contain the words in a usual way[11] To tell the compiler that you need `t` to point to some memory block in the heap, you may declare it in `main` as

[8]Save your code into a file called `reallocWords.c`.

[9]Just after the `#include` lines.

[10]`p` is the address of the memory block to be increased or reduced.

[11]E.g by writing

`char **t;`

in `main`

```
char **t = malloc(sizeof(char**));
```

Also, as you do not want to store the whole user's input in a single long string[12] you need `t` to be a string array, and not just a pointer array. Each element of `t`, `t[i]`, $i = 0, 1, \ldots$, should point to a previously allocated block of memory that is large enough to contain the ith word. When `getWord` returns the buffer, `buf`, contains a new word. To save it into `t` you need to

- increase the length of `t` using
  ```
  t = realloc(t, newSize);
  ```
  where `newSize` is the number of words read so far times the size of a pointer to `char`
- allocate some memory to store the new word using
  ```
  t[index] = malloc((lengthWord + 1)* sizeof(char));
  ```
  where `index` is the the index associated with the new word and `lengthWord` is the length of the word.[13] Note that you should add 1 to the each word's length for the termination character, `\0`, that should be included at the end of each word before saving it into `t`.

Once you have allocated the block of memory to host the new word, you can copy the content of the buffer into the block. To copy a string into another you may use

```
int copyString(char *in, char *out){
        int i = 0;
        while (in[i] != '\0'){
                out[i] = in[i];
                i++;
        }
        out[i]='\0';
        return i;
}
```

Finally, use `free` to free the string array before exiting, if you want to avoid memory leaking and other error or warning messages from Valgrind.

Template

```
int main(){
        char buf[...];
        int end = ...;
        int count = ...;
        char **t = malloc(...);
        while (...){
                int j = getWord(...);
                if (j > 0){
                        buf[j]= ...;
                        t = realloc(...);
                        t[count] = malloc(...);
                        copyString(...);
                        count = ...;
                }
        }
        printWords(...);
        for (...){
                free(...);
        }
```

[12]As you did in Week 3.
[13]You can obtain `lengthWord` with `stringLength` as in Section 1.

```
        free(...);
        return 0;
}
```

Example A run of the program with user input

`one Two three Four five Six seven Eight nine Ten`

should produce the following output

```
1-th word: one (3)
2-th word: Two (3)
3-th word: three (5)
4-th word: Four (4)
5-th word: five (4)
6-th word: Six (3)
7-th word: seven (5)
8-th word: Eight (5)
9-th word: nine (4)
10-th word: Ten (3)
```

If you execute your program with Valgrind (and the same `stdin` input as above), you should not see any error or leaking messages, i.e. Valgrind should print something similar to

```
==3195255==
==3195255== HEAP SUMMARY:
==3195255==     in use at exit: 0 bytes in 0 blocks
==3195255==   total heap usage: 23 allocs, 23 frees, 2,545 bytes allocated
==3195255==
==3195255== All heap blocks were freed -- no leaks are possible
==3195255==
==3195255== For lists of detected and suppressed errors, rerun with: -s
==3195255== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Note This week's quiz will ask you to run the program you wrote in this section. Before proceeding, be sure that your code reproduces exactly the output shown above when you use

`one Two three Four five Six seven Eight nine Ten`

as `stdin` input.

3 Linked list

In this section, you will store the input words into a linked list.[14] Also in this case you will need to fix the maximum length of a word to avoid overflow problems.

Each node should be a `struct` defined as

```
struct node {
    int length;
    char word[MAXLENGTH];
    struct node *next;
};
```

where the `length` member will store the string length, the `word` member the input word, and the `next` member the pointer to the next node.

Have a look at the C code in C example 6.2 to see how you can define and allocate memory for each new node in a linked list. In C example 6.2, you can also see how to print and free a linked list.

[14]Save your code into a file called `linkedWords.c`.

We suggest you modify the code in C example 6.2 so that the linked list contains words from the user's input instead of a series of integers. To parse the user's input and split it into single words you can use part of the code you have written in the Section 2.

Structure your code so that all operations are performed by the following functions

- `int stringLength(char *s)`, to compute the length of a string (see Sections 1 and 2)
- `int copyString(char *in, char *out)` to copy the content of the `in` argument into the `out` argument (see Section 2)
- `int getWord(char *buf, int *end, int maxLength)` to copy a single word from `stdin` into the `buf` argument and set the `end` argument to 1 at the end of the `stdin` input (see Section 2)
- `int printList(struct node *head, int n)` to print the content of the list from its nth node, which is the last that you have added and is pointed by the `head` argument
- `int freeList(struct node *head)` to free the memory allocated in heap for each node, starting from the last node that you have added and is pointed by the `head` argument

To implement the last two functions, you can reuse the code of C example 6.2.[15]

Template As `main`, you can use the following template

```
int main(){
        char buf[MAXLENGTH];
        struct node *head = ...;
        struct node *cur = ...;
        int end = ...;
        int count = ...;
        while (...){
                int j = getWord(...);
                if (j > 0){
                        buf[j]=...;
                        cur = malloc(...);
                        cur->next = ...;
                        copyString(...);
                        cur->length = stringLength(...);
                        head = ...;
                        count = ...;
                }
        }
        int iPrint = printList(...);
        int iFree = freeList(...);
        printf("(count, iPrint, iFree)=(%d, %d, %d)\n", count, iPrint, iFree);
        return 0;
}
```

Example A run of the program with user input

`one Two three Four five Six seven Eight nine Ten`

should produce the following output

```
one Two three Four five Six seven Eight nine Ten
10-th node: Ten (3)
9-th node: nine (4)
8-th node: Eight (5)
```

[15]Try to use the indicated arguments and return values.

```
7-th node: seven (5)
6-th node: Six (3)
5-th node: five (4)
4-th node: Four (4)
3-th node: three (5)
2-th node: Two (3)
1-th node: one (3)
(count, iPrint, iFree)=(10, 10, 10)
```

If you execute your program with Valgrind (and the same `stdin` input as above), you should not see any error or leaking message, i.e. Valgrind should print something similar to

```
==3189409== HEAP SUMMARY:
==3189409==     in use at exit: 0 bytes in 0 blocks
==3189409==   total heap usage: 12 allocs, 12 frees, 3,168 bytes allocated
==3189409==
==3189409== All heap blocks were freed -- no leaks are possible
==3189409==
==3189409== For lists of detected and suppressed errors, rerun with: -s
==3189409== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```