CS2106: Introduction to Operating Systems

Lab Assignment 1 (A1)
C Programming

Important:

- **The deadline of submission through LumiNUS: Saturday, Feb 8th, 3pm**
- The total weightage is 3%:
    - Exercise 1: 1 % **[Lab demo exercise]**
    - Exercise 2: 1 %
    - Exercise 3: 1 %
- **You must ensure the exercises work properly on the lab machines (i.e., Linux on x86)**

This lab has 3 exercises. You will demonstrate your program for exercise 1 to your tutor during lab session. For exercises 2 and 3, you need to submit the relevant file(s) to LumiNUS (submission instruction given in Appendix). Please ensure that your exercises work on a Linux machine

Exercise 1 [Lab Demo Exercise]

Code for this exercise can be found in **ex1** folder.

In this exercise, you will implement functions to support circular doubly linked list operations.
The struct for node has already been defined for you in **node.h** as below:

```
typedef struct NODE {
  struct NODE *prev;
  struct NODE *next;
  char *word;
} node;
```

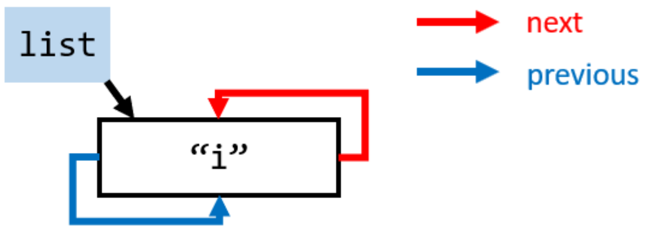
It is assumed that **word** is null-terminated, has at most 10 characters (excluding the null terminator), and only consists of alphabets and/or numbers. During your own testing, there is no need to use strings that violate these constraints.

The struct for list has also been defined for you in **node.h** as below:

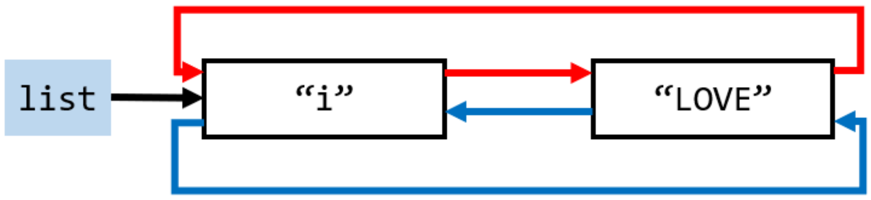
```
typedef struct {
  node *head;
} list;
```

An empty list will have a null pointer for **head**. For any non-empty list, we index the nodes in it starting from **head** (which has index 0). The next page shows some examples of possible circular doubly linked lists using this representation.

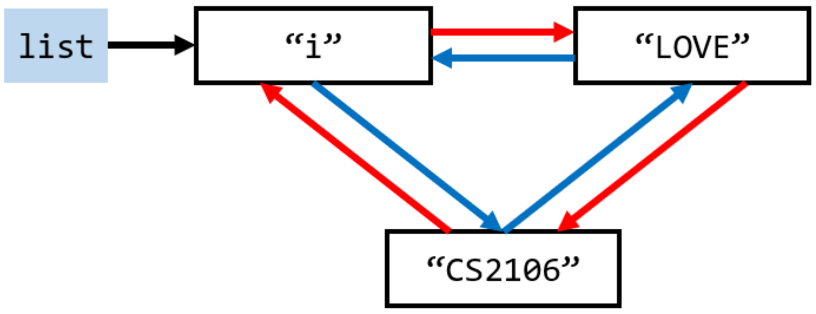
Initial list with only one node containing **"i"**:



List after inserting a node containing **"LOVE"** word after the node at index 0.



List after inserting a node containing **"CS2106"** word before the node at index 0.



In the final list, nodes containing **"i"**, **"LOVE"**, **"CS2106"** have the indices 0, 1, 2 respectively. As this is a doubly linked list, we can also index the list with negative indices. Then the nodes containing **"i"**, **"LOVE"**, **"CS2106"** have indices 0, -2, -1 respectively.
Also, this is a circular linked list, so we can wrap around the list to handle "out-of-bound" indices. For example, node containing **"i"** can be said to have indices 0, 3, 6 and so on; node containing **"LOVE"** can be said to have indices -2, -5, -8 and so on.

You need to implement the following 5 functions in **node.c**:

a) `void insert_node_before(list *lst, int index, char *word)`
   This function inserts a new node that contains a **word** before the node currently at **index** of **lst**. If **lst** is empty, then **lst->head** should point to the inserted node. The caller of this function will not **free**[1] memory allocated for **word**, so you can copy over the pointer instead of the characters one-by-one. You should use **malloc**[2] to allocate memory to create the new node. Note that when a node is inserted before **lst->head** (assuming non-empty list), **lst->head** will not change.

b) `void insert_node_after(list *lst, int index, char *word)`

[1] Sample usage of **free** can be found in appendix.
[2] Sample usage of **malloc** can be found in appendix.

This function behaves in the same way as **`insert_node_before`**, except that new node is inserted after the node at **`index`** of **`lst`**. (Hint: can you re-use **`insert_node_before`** to implement this function?) Note that when a node is inserted after **`lst->head->prev`** (assuming non-empty list), **`lst->head`** will not change.

c) `char *list_to_sentence(list *lst)`

This function returns a sentence containing all words in **`lst`** organized in the following manner:

a) If the character is a number, it remains unchanged.
b) If the character is an alphabet and the first letter of the sentence, copy the capitalized letter to returned sentence; otherwise, copy the lowercase letter (even if it is in upper case originally).
c) The sentence must end with a full stop and there must be exactly one blank space character between printable characters of neighboring words.
d) The sentence must be null terminated.
e) The list is traversed and the word is collected by repeatedly visiting the next node (if any) until it goes back the starting node.
f) It is guaranteed that **`lst`** is a non-empty list.

In the final list shown in the example above, if **`lst->head`** points to A, then **"`I love cs2106.`"** should be returned. You may find **`strlen`** from **`string.h`** helpful for this function. The caller of this function will take the responsibility to **`free char *`** afterwards. You don't have to be worried about memory management for the returned **`char *`**.

d) `void delete_node(list *lst, int index)`

This function deletes the node at **`index`** in **`lst`**. You should use **`free`** to release memory allocated for the deleted node and **`word`** of the node. **`lst`** here is guaranteed to be a non-empty list. If head gets deleted, next node (if any) will become the new head.

e) `void delete_list(list *lst)`

This function deletes all nodes in **`lst`**. In other words, memory allocated for the nodes in **`lst`** should all be freed after calling this function using **`free`**. **`lst`** here is guaranteed to be a non-empty list.

It is guaranteed that **`lst`** parameter will not be a null pointer and input validation is not required for any function. **It is also guaranteed that the input file will end with a command to delete the list.**

If you want to implement additional helper functions, include the function prototypes in **`node.c`** itself. Changes in **`node.h`** or **`ex1.c`** will be overwritten because we will be using our own copies during testing (changes modified in your copies will not be reflected during testing).

To help test your program, a sample test case has been provided to you in **`1_1.in`** and the expected output is in **`1_1.ans`**. Each line in **`1_1.in`** starts with a number, which indicates the function being called by that line. (The number to function mapping is reflected with **`#define`**s in the top of **`ex1.c`**). For example, if a line starts with 1, then this line is calling **`insert_node_before`**; if a line starts with 4, then this line is calling **`delete_node`**. When the called function has more arguments, the same line will have more numbers or characters to be read in (space-separated) as further inputs. For example, a line with **"`1 2 abc`"** is inserting a new node before the node at index 2 (0-based index) of the list and this newly inserted node contains **"`abc`"** as its word. The parsing of inputs has already been done for you (**`run`** in **`ex1.c`**).

After you have finished implementation, use the following command to compile your code:

```
make
```

This will produce the compiled binary files **`node.o`** and **`ex1`**. (This comes from **`Makefile`** in the directory. You can read up more about it if you are interested)

To execute your program and let it read inputs from **`1_1.in`**, you can use the following command:

```
./ex1   <  1_1.in  >  1_1.out  2> 1_1.error_log
```

This command will redirect the standard input (stdin) to file 1_1.in, standard output (stdout) to 1_1.out, and the standard error (stderr) output to 1_1.error_log. Normally, the error log file should be empty.

Then you can compare **`1_1.out`** with the expected answer provided in **`1_1.ans`** using the following command:

```
diff 1_1.ans 1_1.out
```

This command outputs the difference(s) between the 2 specified files.

Each time before you compile the code again, run **`make  clean`** to clear the previously generated binary files.

For this lab and future labs where sample test case(s) may be provided, passing the sample test case does not guarantee full marks for the lab. You should write more test cases on your own and test further to ensure your program works correctly.

Exercise 2.1

Code for this exercise can be found in **`ex2`** folder.

In this exercise, you will implement functions to support simple matrix operations using pointers. The struct for matrix has already been defined for you in **`matrix.h`** as below:

```
typedef struct {
  int **data;
  int num_rows;
  int num_cols;
} matrix;
```

You need to implement the following functions in **`matrix.c`**:

a) `matrix *create_matrix(int num_rows, int num_cols)`
This function creates a zero matrix (aka, all cells in the matrix are zero) with the specified dimensions. It is up to you to decide how to allocate memory to store values in **`int **data`**, but you must make sure matrix is stored in row-major order. That is, **`data[r-1][c-1]`** contains the element of the matrix at **`r`**th row and **`c`**th column (the first row/column is row/column zero).

b) `void add_row(matrix *mat, int *row)`[3]
This function adds a row to the bottom of the matrix, i.e., the added row will become the last row. **`row`** is guaranteed to be valid and have the same number of elements as specified by **`mat->num_cols`**. Note that **`row`** will be **`free`**d by caller after calling this function, so you need to copy over the element one-by-one. You may find **`realloc`** from **`stdlib.h`** useful.

c) `void add_col(matrix *mat, int *col)`
This function adds a new column to the right of the matrix, aka, the added column will become the rightmost column. **`col`** is guaranteed to be valid and have the same number elements as specified by **`mat->num_rows`**. Note that **`col`** will be freed by caller after calling this function, so you need to copy over the element one-by-one.

d) `void increment(matrix *mat, int num)`
This function increments each value in the matrix by **`num`**.

e) `void scalar_multiply(matrix *mat, int num)`
This function multiplies each value in the matrix by **`num`**.

f) `void scalar_divide(matrix *mat, int num)`
This function divides each element in the matrix by **`num`**. For simplicity, you can just use integer division, aka, you can just use **`/`** operator. It is guaranteed that **`num`** will not be 0.

g) `void scalar_power(matrix *mat, int num)`

[3] You may find **`realloc`** and/or **`memcpy`** useful for **`add_row`** and **`add_col`** but **`malloc`** and **`free`** are enough. However, we won't restrict the use of **`realloc`** and **`memcpy`** if you want to use them.

This function raises each element in the matrix to the power of **num**. You need to implement a function to calculate power.

h) `void delete_matrix(matrix *mat)`
This function deletes **mat**. You should use **free** to release all memory allocated for **mat**.

You can assume the inputs and subsequent operations will not lead to integer overflow and there is no need for input validation. **It is guaranteed that the input file will start with a command to create the initial matrix and end with a command to delete the matrix.**

If you wish to implement additional helper functions, include the function prototypes in **matrix.c** itself. Changes in **matrix.h**, **ex2.h** and **ex2.c** will be overwritten because we will be using our own copies during testing (changes modified in your copies will not be reflected during testing).

To help test your program, we have provided a sample test case in **2_1.in** and the expected output is in **2_1.ans**. To test, use the following commands:

```
make
./ex2 < 2_1.in > 2_1.out 2> 2_1.error_log
diff 2_1.out 2_1.ans
```

To remove the generated binary files, use **make clean**.

The first line of **2_1.in** has 2 numbers, R and C, which specify the dimensions of initial matrix. R is the number of rows and C is the number of columns. The following lines all begin with a number that indicates the function to call for that line (the number to function mapping is reflected with **#define**s in the top of **ex2.h**). Depending on which function is called, there may or may not be more numbers on the same line to be read in as arguments.
The function to parse these texts and call the respective functions have already been defined for you (**run** in **ex2.c**).

Again, passing the sample test case does not guarantee full marks. You should write more test cases on your own and test further to ensure your program works correctly.

Exercise 2.2

In exercise 2.1, you have implemented several functions that perform an element-wise operation on the input matrix. You should notice that all such functions (d)-g) from the list in ex2 above) look largely identical. This is where we can use function pointers to simplify our code.

Suppose we have defined a function with the following implementation:

```
void increment_single(int *x, int y)
{
  (*x) += y;
}
```

Then a pointer to this function can be declared and initialized as follows:

```
void (*increment_single_ptr)(int *, int) = increment_single;
```

Note that pointer **increment_single_ptr** is initialized to contain the address in memory where the compiled version of function **increment_single** is located. Hence, we can dereference **increment_single_ptr** to invoke the underlying function **increment_single**.

Hence, the following 2 lines of codes achieve the same effect:

```
increment_single(some_int_ptr, 2);
(*increment_single_ptr)(some_int_ptr, 2);
```

With that in mind, implement the following function in **matrix.c**:

```
void element_wise_op(matrix *mat, int num, void (*op_ptr)(int *, int))
```

This function performs element-wise operation specified by **op_ptr** on each value in **mat**.
For example, **element_wise_op(mat, num, increment_single_ptr)** should do the same job as **increment(mat_ptr, num)** from exercise 2, where **mat** is an **matrix ***, **num** is an integer and **increment_single_ptr** is a function pointer defined as above.

You don't have to implement functions like **increment_single** for this task but it may be helpful if you wish to test your **element_wise_op**.

Again, you can assume the inputs will not cause overflow and there is no need for input validation.

Exercise 3

Code for this exercise can be found in **`ex3`** folder. Please copy over your completed **`matrix.c`** from exercise 2 to this folder.

In exercise 2, the function **`run`** (already implemented for you) parses the input stream and uses if-else statements to decide which function to call. An alternative is to use an array of function pointers. We can index on the array (as each function is mapped to a number) to retrieve the corresponding function pointer and then invoke the correct function.

However, there is an issue with using an array of function pointers that directly point to the functions implemented in exercise 2. Arrays can only contain elements of the same type. When it is an array of function pointers, then all the underlying functions must share the same function signature. However, this is not true for functions in exercise 2.

A workaround is to wrap each of the functions in another function referred to as function wrapper. Although functions in exercise 2 have different signatures, we can create a function wrapper for each of them and ensure the function wrappers share the same signature. Then we can create an array of function pointers that point to those wrappers.

For the purpose of this exercise, refrain from declaring an array of generic function pointers that accepts all function signatures.

Following the guidance provided above, implement the following function in **`ex3.c`**, using an array of function pointers to function wrappers.

`void my_run()`

This function should be able to replace the given **`run`** function in **`ex2.c`**. It invokes function calls according to what the input stream specified.
Hint: You will need to implement function wrappers as well. But you are free to decide what signatures these wrappers should have.

You can assume the inputs and subsequent operations will not lead to integer overflow and there is no need for input validation. **It is guaranteed that all input files start with a command to create the initial matrix and end with a command to delete the matrix.**

For this question, changes in **`matrix.h`**, **`ex3.h`**, **`ex3_runner.c`** will be overwritten because we will be using our own copies during testing (changes modified in your copies will not be reflected during testing). You can change other files.

Appendix

Sample usage of `malloc`

`malloc` is used to allocate memory on heap. Hence, we need to specify the size of memory to be allocated to our variable. This can be achieved by calling **`sizeof(<variable's type>)`**. Upon success, **`malloc`** returns a pointer to the beginning of the memory region allocated. Note that as **`malloc`** simply allocates the specified size of memory, it is not aware of the underlying data type. Hence it is a good practice to cast the returned pointer to the correct pointer type like below:

```
int *a_ptr = (int *) malloc(sizeof(int) * 4);
```

The code above allocates memory to store 4 integers, casts returned pointer to **`int *`**, then assign it to **`a_ptr`** variable. Besides **`malloc`**, **`calloc`** is also used to allocate memory. You can find out more about it if you are interested.

Sample usage of `free`

To release the memory allocated for **`a_ptr`** (**`malloc`**ed as above), we simply call **`free(a_ptr)`**. We do not need to explicitly specify the size of memory to be released because **`malloc`** already stores accounting information about the amount of memory allocated to **`a_ptr`**. So **`free`** just reads that information to determine the size of memory to release.

However, when we have double pointers or pointers stored in struct, we cannot simply free the "outermost" pointer, because the accounting information for this pointer only records how much memory is used for this pointer itself, not the memory (nor the value) of the other pointers "nested" inside. Hence, to release memory properly, we need to free them from "inside out".

Submission through LumiNUS

Zip the following files with the following folder structure as E0123456.zip (**replace E0123456 with your NUSNET id, NOT your student no A012…B, and use capital 'E' as prefix**):

```
ex2/
      matrix.c
ex3/
      ex3.c
      matrix.c
```

Do not add additional folder structure during zipping, e.g. do not place the above in a "**`lab1/`**" subfolder etc. Upload the zip file to the "Student Submission Lab 1" folder on LumiNUS. Note the deadline for the submission is **Saturday, Feb 8th, 3:00pm.** Please ensure you follow the instructions carefully (output format, how to zip the files etc). Deviations will cause unnecessary delays in the marking of your submission.

Questions?

Please ask your questions on the relevant LumiNUS forum.