CS2106: Introduction to Operating Systems

Lab Assignment 1 (A1)
Advanced C Programming

Important:

- **The deadline of submission through LumiNUS: Sat, 7 Oct, 2pm**
- The total weightage is 5%:
  - Exercise 1: 1 % **[Lab demo exercise]**
  - Exercise 2: 2 %
  - Exercise 3: 2 %
- **You must ensure the exercises work properly on the lab machines (i.e. Linux on x86)**

Section 1. General Information

Here are some simple guidelines that will come in handy for all future labs.

1.1. Lab Assignment Duration & Lab Demonstration

Each lab assignment spans about **two weeks** and consists of multiple exercises. One of the exercises is chosen to be the "lab demo exercise" which you need to demonstrate to your lab TA. For example, in this lab, you need to demo exercise 1. The demonstration serves as a way to "kick start" your effort as well as a way to mark your **lab attendance**. You are **strongly encouraged to** finish the demo exercise before coming to the lab.

The remaining lab exercises are usually quite intensive. Do not expect to finish the exercise during the allocated lab session. The main purpose of the lab session is to clarify doubts with the lab TAs and ensure your exercises work properly under Linux.

1.2. Setting up the exercise(s)

For every lab, we will release two files in LumiNUS "Labs" files:

- **`labX.pdf`**: A document to describe the lab question, including the specification and the expected output for all the exercises.
- **`labX.tar.gz`**: An archive for setting up the directories and skeleton files given for the lab.

For unpacking the archive:

1. Copy the archive **`labX.tar.gz`** into your account.
2. Enter the following command:
   `tar -zxvf labX.tar.gz`
   `OR gunzip –c labX.tar.gz | tar xvf –`
   Remember to replace the ***X*** with the actual lab number.

3. The above command should setup the files in the following structure:

```
ex1/                subdirectory for exercise 1
      ex1.c                 skeleton file for exercise 1
      testY.in      sample test inputs, Y= 1, 2, 3, ...
      testZ.out     sample outputs, Z = 1, 2, 3, …
ex2/
      ...           Similar to ex1
ex3/                Similar to ex1
...
```

1.3. Testing using the sample test cases

For most of the exercises, some number of sample input/output are given. The sample input/output are usually simple text file, which you can view them using any editor or pager program.

You can opt to manually type in the sample input and check the output with the standard answer. However, a more efficient and less tedious way is to make use of **redirection** in Unix.

Let us assume that you have produced the executable **`answer.exe`** for a particular exercise. You can make use of the sample test case with the input redirection:

```
answer.exe < test1.in
```

The above "tricks" the executable **`answer.exe`** to read from **`test1.in`** **as if it was the keyboard input**. The output is shown on the screen, which you can manually check with **`test1.out`**.

Similarly, make use of output redirection to store the output in a file to facilitate comparison.

```
answer.exe < test1.in > myOut1.txt
```

The effect of the above command is:

- Take **`test1.in`** as if it is the standard input device
- Store all output to **`myOut1.txt`** as if it is the standard output device
    - Obviously, any other filename can be used
    - Just be careful not to overwrite an existing file

With the output store in **`myOut1.txt`**, you can utilize the **`diff`** command in Linux to compare two files easily:

```
diff test1.out myOut1.txt
```

which compares the output produced by your program with the sample output. Do a "**`man diff`**" to understand more about the result you see on screen.

You are **strongly encouraged** to check the output in this fashion. By making sure your answer follows the standard answer (especially the format), you free up more time for the Lab TA to give better comments and feedback on your program.

Note that we will use **additional test cases** during marking to further stress test your submission. It is also part of your training to come up with "killer tests" for your own code. ☺

Section 2. Exercises in Lab 1

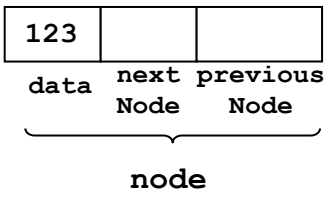
There are **three exercises** in this lab. Although the main motivation for this lab is to familiarize you with some advanced aspects of C programming, the **techniques** used in these exercises are quite commonly used in OS related topics.

For this lab, we will "simulate" a file system and events happening on the folders/files in this system. Specifically, exercises 1 and 2 implement a 2-level list to simulate storing the folders and sub-folders in a file system. Exercise 3 takes in different events and applies them on the list of folders.

2.1 Exercise 1 [Lab Demo Exercise]

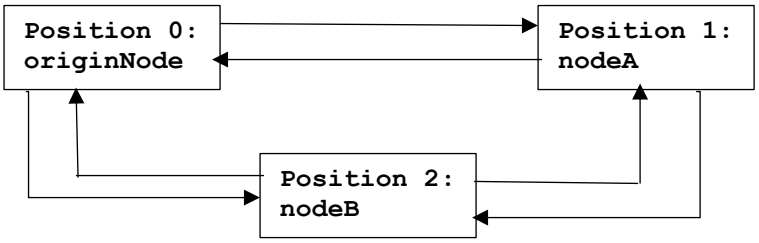
Let's implement a list to store the files structure located in a directory. For exercise 1, we store the list of files located in a folder using a **circular double linked list.** For simplicity, the data contained in the nodes is an integer.

```
typedef struct NODE{
    int data;
    struct NODE* previousNode;
    struct NODE* nextNode;
} node;
```



Assume a list with one node, **originNode,** has been created for you in advance. Both previousNode and nextNode are pointers to the same originNode. If another node is inserted, the two pointers would point to that new node instead. Positions in the list are relative to the **originNode** (position 0) and are only positive.

Below is an illustrated example:




```
originNode->previousNode == nodeB;
originNode->nextNode == nodeA;
```

```
nodeA->nextNode == nodeB;
nodeA->previousNode == originNode;

nodeB->nextNode == originNode;
nodeB->previousNode == nodeA;
```

NOTE: If a new node is inserted after nodeA, nodeA's position remains unchanged. The new node takes position 2, and nodeB takes position 3.

This exercise requires you to write **four functions**:

`void insertNodePrevious (int position, int value, node* originNode);`
This function inserts a node with data **`value`** before the node at position **`position`** in list **`originNode`**. Allocate memory for the new node using `malloc()` function. `Position` takes values greater or equal to 0, and wraps around the circular list. The position is relative to the **originNode** as seen in the illustrated example before.

`void insertNodeNext(int position, int value, node* originNode);`
This function inserts a node with data **`value`** after the node at position **`position`** in list **`originNode`**. Allocate memory for the new node using `malloc()` function. `Position` takes values greater or equal to 0, and wraps around the circular list.

`void deleteNode(int position, node* originNode);`
This function removes node at position **`position`** from list **`originNode`**. Additionally it deallocates the corresponding memory by using `free()` function.

`void deleteList(node* originNode);`
This function uses `free()` to delete all nodes in list **`originNode`** list, excluding the initial node.

You are allowed to write other functions if needed.

The program should:

a. Read in an instruction and position.
b. If instruction is -1, read in data and insert it before the node at position in the circular double linked list.
c. Else if instruction is 1, read in data and insert it after the node at position in the circular double linked list.
d. Else if instruction is 0, delete node at position.
e. Go to step a, until user terminates input by pressing **`Ctrl-D`**
    - **`Ctrl-D`** is the "end-of-file" signal
f. Print out the whole list

g. Delete the list excluding the originNode
h. Print out the list (should only include originNode)

Note: Only valid positions are inserted and originNode is not deleted:

1. **Positions given as input are positive or 0. If the position is greater than the number of nodes in the list, the insertion is done by continuing to traverse and count in the circular list (wrap around).**
2. **`deleteNode()` is not used on originNode. The position given as input is positive (>0) and less than the number of nodes in the circular list (no wrap around).**

The skeleton file **`ex1.c`** has the following:

- The input/output code is already written.
- A useful function **`printList()`** is written to print out a correctly constructed linked list.

Sample Input 1:

```
1 0 10      //Insert a node with value 10 after node at position 0.
-1 0 5      //Insert a node with value 5 before node at position 0.
1 2 7       //Insert a node with value 7 after node at position 2.
-1 3 2      //Insert a node with value 2 before node at position 3.
[Ctrl-D]
```

Sample Output 1:

```
Printing clockwise:
[Pos 0:0]
   |
   v
[Pos 1:10]
   |
   v
[Pos 2:5]
   |
   v
[Pos 3:2]
   |
   v
[Pos 4:7]
   |
   v
[Pos 0:0]
Printing counter-clockwise:
[Pos 0:0]
   |
   v
[Pos 4:7]
   |
   v
[Pos 3:2]
   |
```

```
   v
[Pos 2:5]
   |
   v
[Pos 1:10]
   |
   v
[Pos 0:0]
Circular List after delete
Printing clockwise:
[Pos 0:0]
   |
   v
[Pos 0:0]
Printing counter-clockwise:
[Pos 0:0]
   |
   v
[Pos 0:0]
```

Sample Input 2:

```
1 0 10     //Insert a node with data 10 after node at position 0.
-1 0 5     //Insert a node with data 5 before node at position 0.
1 5 7      //Insert a node with data 7 after node at position 5. (wrap around)
-1 3 2     //Insert a node with data 2 before node at position 3.
0 1       //Delete node at position 1.
0 2       //Delete node at position 2.
[Ctrl-D]
```

Sample Output 2:

```
Printing clockwise:
[Pos 0:0]
   |
   v
[Pos 1:5]
   |
   v
[Pos 2:7]
   |
   v
[Pos 0:0]
Printing counter-clockwise:
[Pos 0:0]
   |
   v
```

```
[Pos 2:7]
   |
   v
[Pos 1:5]
   |
   v
[Pos 0:0]
Circular List after delete
Printing clockwise:
[Pos 0:0]
   |
   v
[Pos 0:0]
Printing counter-clockwise:
[Pos 0:0]
   |
   v
[Pos 0:0]
```

Important Note:

The correctness of your program cannot be entirely verified just by the output. For example, the linked list can be freed in the wrong way (*hint!*) but still seems to work. So, as a programmer, you need to scrutinize the code instead of just relying on the output.

This marks an important progression of your skill and understanding. Of course, it is also a source of major pain for you as there is now no "simple way" to ensure your code is 100% correct. Just a friendly warning: most, if not all, lab exercises in CS2106 share this characteristic.

2.2 Exercise 2

Now you can use the list from exercise 1 to create a list of lists to simulate a 2-levels folder structure in an operating system (this is like a tree structure with two levels). For exercise 2, we expand on the previous exercise and implement a **circular double linked list of linked lists**!

```
typedef struct SUBNODE{
    int data;
    struct SUBNODE* nextSubNode;
} subNode;

typedef struct NODE{
    int data;
    struct NODE* previousNode;
    struct NODE* nextNode;
    subNode* subNodehead;
} node;
```

Each node now hosts a list of **subNodes.** The list of subNodes is represented using a linked list. Similar to exercise 1, the subNodes in the linked list are indexed starting with position 0.

You need to write at least **one function** for this exercise:

`void insertSubNode(int position, int subPosition, int value, node* originNode);`
This function inserts a subNode with data **`value`** at position **`subPosition.`** The subNode is part of the linked list located at position **`position`** in list **`originNode`**. `Malloc()` is used to dynamically allocate memory to the subNode. Cases to consider: 1. If subNode list of a node is empty, adding a subnode to subPosition 0 makes that subNode the head of the list. 2. If the subNode list has 10 subNodes, adding a subnode at subPosition 5 inserts the new node at subPosition 5 and shifts all subNodes after 5 by one subPosition. 3. If the subNode list has 10 subNodes (ie subPosition 0 – 9 occupied), subPosition 10 is a valid position to add a subNode. However, subposition 12 is not a valid position.

Additionally, you need to create new functions for exercise 1 to cater to this new data structure. Use new names for the functions that you create. When inserting a node in the circular double linked list, the subNode linked list is NULL. (*Hint!* `deleteList()` should also `free()` all subNodes, and `deleteNode()` should delete the subNodeList.)

Note: All arguments given as position/subPosition are valid values and can be added to the lists.

The program should:

a. Read in an instruction and position.
b. If instruction is -1, scan for data and insert previous to node at position.
c. Else if instruction is 1, scan for data and insert next to the node at position.
d. Else if instruction is 0, delete node at position.
e. **Else if instruction is 2, scan for data and subPosition and execute `insertSubNode()`.**
f. Go to step a, until user terminates input by pressing **`Ctrl-D`**
    - **`Ctrl-D`** is the "end-of-file" signal
g. Print out the whole list
h. Delete the list excluding the originNode
    - But do delete the subNodeList of originNode
i. Print out the list (should only include originNode)

Sample Input 3:

```
1 0 5      //Insert a node with data 5 after node at position 0.
1 1 10     //Insert a node with data 10 after node at position 1.
2 0 0 1    //For node at position 0, insert subnode with data 1 at subPosition 0.
2 0 1 10   //For node at position 0, insert subnode with data 10 at subPosition 1.
2 1 0 2    //For node at position 1, insert subnode with data 2 at subPosition 0.
2 1 0 20   //For node at position 1, insert subnode with data 10 at subPosition 0.
[Ctrl-D]
```

Sample Output 3:

```
Printing clockwise:
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
   |
   v
[Pos 1:5]->[subNode pos 0:20]->[subNode pos 1:2]
   |
   v
[Pos 2:10]
   |
   v
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
Printing counter-clockwise:
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
   |
   v
[Pos 2:10]
   |
   v
[Pos 1:5]->[subNode pos 0:20]->[subNode pos 1:2]
   |
   v
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
```

```
Circular List after delete
Printing clockwise:
[Pos 0:0]
   |
   v
[Pos 0:0]
Printing counter-clockwise:
[Pos 0:0]
   |
   v
[Pos 0:0]
```

Sample Input 4:

```
1 0 5      //Insert a node with data 5 after node at position 0.
1 1 10     //Insert a node with data 10 after node at position 1.
2 0 0 1    //For node at position 0, insert subnode with data 1 at subPosition 0.
2 0 1 10   //For node at position 0, insert subnode with data 10 at subPosition 1.
2 1 0 2    //For node at position 1, insert subnode with data 2 at subPosition 0.
2 1 0 20   //For node at position 1, insert subnode with data 10 at subPosition 0.
0 1        //Delete node 1.
[Ctrl-D]
```

Sample Output 4:

```
Printing clockwise:
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
   |
   v
[Pos 1:10]
   |
   v
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
Printing counter-clockwise:
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
   |
   v
[Pos 1:10]
   |
   v
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
Circular List after delete
Printing clockwise:
[Pos 0:0]
   |
   v
[Pos 0:0]
Printing counter-clockwise:
```

```
[Pos 0:0]
   |
   v
[Pos 0:0]
```

Important Note:

Similar to exercise 1, the correctness of your program cannot be entirely verified just by the output. For example, the linked list can be freed in the wrong way (*hint!*)

2.3 Exercise 3

The last exercise is on **function pointer**. Unlike normal pointer, which points to memory location for **data storage,** a function pointer **points to a piece of code (function)**. By dereferencing a function pointer, we **invoke the function** that is referred by that pointer. This technique is commonly used in **system call / interrupt handlers**.

Suppose we have two functions:

```
void f (int x);

void g (int y);
```

They are considered as the same "type" of functions because **the number and datatype of parameters, as well as the return type is the same**. (More accurately, we say the **function signature** of the two functions are the same).

In C, it is possible to define a **function pointer** to refer to a function. For example:

```
void (*fptr) ( int );
```

To understand this declaration, imagine if you replace **`(*fptr)`** as **`F`**, then you have:

```
void F( int );
```

So, **`F`** is "a function that takes an integer as input, and return nothing (void)".

Now, since **`(*fptr)`** is **`F`**, **`fptr`** is "**a pointer to** a function that takes an integer as input, and return nothing (void)". Let's use this concept to create a group of functions that can handle some events.

Exercise 3 is an extension of exercise 1. You need to create an **array of function pointers**! Each element in the array (function) is called to handle different events on the circular double linked list. Hence, your program can only carry out any of the functions in exercise 1 by dereferencing those function pointers! (ie: `insertNodeNext()`, `deleteNode()`, `insertSubNode()`, `etc`) Additionally, you can have call one more function, `collapseSubNode`.

You need to write at least **one function + input processing** for this exercise:

`void collapseSubNodes (int position, node* targetNode)`
This function collapses the linked list from position **`position`** at the list **`targetNode`** by adding up all the data values of the subList of subNodes of that node. The subList is then properly deallocated with `free()` and deleted. The sum os the subList elements is added to the value of the node at **`position`** in the list **`targetNode`**.

The program should:

a. Read in a position and instruction.
b. If instruction is -1, scan for data and insert before node at position.
c. Else if instruction is 1, scan for data and insert next to the node at position.
d. Else if instruction is 0, delete node at position.
e. Else if instruction is 2, scan for data and subPosition and execute `insertSubNode()`.
f. Else if instruction is 3, execute `collapseSubNodes()` for node at position.
g. Go to step a, until user terminates input by pressing **`Ctrl-D`**
    - **`Ctrl-D`** is the "end-of-file" signal
h. Print out the whole list
i. Delete the list excluding the originNode
    - But do delete the subNodeList of originNode
j. Print out the list (should only include originNode)

Note: You will be awarded points only if your functions are called by dereferencing function pointers!

Sample Input 5:

```
1 0 5      //Insert a node with data 5 after node at position 0.
1 1 10     //Insert a node with data 10 after node at position 1.
2 0 0 1    //For node at position 0, insert subnode to subPosition 0 with data 1
2 0 1 10   //For node at position 0, insert subnode to subPosition 1 with data 10
2 1 0 2    //For node at position 1, insert subnode to subPosition 0 with data 2
2 1 0 20   //For node at position 1, insert subnode to subPosition 0 with data 10
3 1        //Collapse node 1.
[Ctrl-D]
```

Sample Output 5:

```
Printing clockwise:
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
   |
   v
[Pos 1:27]
   |
   v
[Pos 2:10]
   |
   v
```

```
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
Printing counter-clockwise:
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
   |
   v
[Pos 2:10]
   |
   v
[Pos 1:27]
   |
   v
[Pos 0:0]->[subNode pos 0:1]->[subNode pos 1:10]
Circular List after delete
Printing clockwise:
[Pos 0:0]
   |
   v
[Pos 0:0]
Printing counter-clockwise:
[Pos 0:0]
   |
   v
[Pos 0:0]
```

For your pondering:

Function pointer, though seems like a weird C language quirks, is actually a very powerful programming technique. Consider the following:

- How can we quickly change the mapping of the functions (say now you want a different function to be used for inserting a node in the linked list of the circular double linked list)?
- How can we modify the functionalities without affecting other part of the code (say you are using this array of functions to handle events, and you want to able to handle additional events with a minimum effort)?
- Is the function pointer approach faster / easier to write than selection statement (think about having more than 100+, how would your code look like if you need to write selection statements)?

This mechanism is available in many other programming languages, albeit with a different name, e.g. JavaScript allows you to bind function to a variable, functional programming languages, e.g. Python allows you to pass function around as a "value" (more formally known as **function as first class citizen**), etc.

Section 3. Submission through LumiNUS

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

a. ex2.c
b. ex3.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 **7 Sep, 2pm.**

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.