Middle East Technical University | Department of Computer Engineering

CENG 140

C Programming

Spring '2019-2020
Take-Home Exam 3

Emre Külah
kulah@ceng.metu.edu.tr
Due date: June 7, 2020, Sunday, 23:59

1 Dining Philosophers

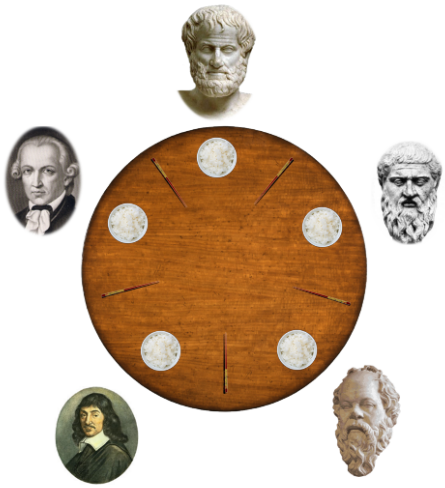

Figure 1: Dining Philosophers

K silent philosophers decided to meet and have a dinner around a round table. They want to eat their favorite meals. This scenario is inspired by the famous synchronization example which you will be familiar with in Operating Systems course. In the scope of this task, we only want to seat them around a table and serve their favorite meals. Also, anyone can decide to leave the table anytime he wants.

2 Structs and Linked List

You will be given two struct definitions to represent philosophers and meals.

```
typedef struct Meal
{
    char *name; \\ name of the meal
    int count; \\ number of servings
} Meal;

typedef struct Philosopher
{
    char *name; \\ name of the philosopher
    char *favorite_meal; \\ name of his favorite meal
    int age; \\ age of the philosopher
    int sitting; \\ stores whether he is sitting around the table or not
} Philosopher;
```

One struct definition for linked list representation.

```
typedef struct Node
{
    void *node;
    struct Node *next;
} Node;
```

3 Tasks

Using these 3 struct definitions, you will be initialize two singly linked lists. One for meals and one for philosophers. These lists will store all of the meals and philosophers. The number of items in these lists will not be changed.

- **void add_meal(Node **meals_head, char *name, int count)** function gets name and count as parameter and creates a node for this meal. Append this node to the end of the linked list given as meals_head.
- **void add_philosopher(Node **philosophers_head, char *name, char *favorite_meal, int age)** function gets name, favorite meal and age as parameter and creates a node for this philosopher. Append this node to the end of the linked list given as philosophers_head.

While you create two linked lists, you are going to place philosophers around the table in the ascending order of their ages. The table will be represented as circular singly linked list, using Node struct.

- **void place_philosophers(Node **table_head, Node *philosophers)** function gets table head and philosophers linked list as parameter and place philosophers. You are going to allocate a new space for each Node but "Node *data" will point the corresponding original philosopher. Figure 2 shows the architecture You are going to update "sitting" member to 1 when corresponding philosopher placed.
- **void remove_philosopher(Node **table_head, int index, int size_of_table)** function gets table head, index of the philosopher to be removed and size of the table. You are going to remove corresponding philosopher from the table. **Be careful when index is 0.**

After you placed the philosophers around the table, you are going to serve their favorite meals.

- **void serve_meals(Node *table, Node *meals)** function gets table and meals list as parameter. Serves each philosopher's favorite meals and reduce the number of that meal from the meals list.

There are 7 main tasks in this homework and two of them are printing functions. These are used to print linked lists and print table.

- **void print_list(Node *list, void (*helper_print)(void *))** function gets a linked list and a reference to a helper function. This helper function takes a "void *" as parameter and prints its content. There are two helper functions for this duty.
  - **void print_meal_node(void *meal)** →
    Name: #name_of_the_meal#, count: #count_of_the_meal#
  - **void print_philosopher_node(void *philosopher)** →
    Name: #name_of_the_philosopher#, favorite meal: #favorite_meal#, age: #age#
  - These helper functions get a void *data, cast it to corresponding type (Meal* or Philosopher*) and then print it content in given format.

  Our Node struct is generic and keeps "void *". For this reason, we have to implement two different helper function for meals and philosophers. You are going to pass these two functions to **print_list** and use them as follows:

```
void print_list(Node *list_head, void (*helper_print)(void *)){
    \* some lines *\
    helper_print(<a node pointer>)
    \* some lines *\
}

print_list(meals, &print_meal_node);
print_list(philosophers, &print_philosopher_node);
```

One of the important functions is printing the table. However, since it is a circular singly linked list, we are going to use a different format.

- **void print_table(Node *table)** gets table as parameter and prints sitting philosophers as following format:
  - $< n^{th} philosopher >< 1^{st} philosopher >< 2^{nd} philosopher >$
  - $< 1^{st} philosopher >< 2^{nd} philosopher >< 3^{rd} philosopher >$
  - $< 2^{nd} philosopher >< 3^{rd} philosopher >< 4^{th} philosopher >$
  - $\vdots$
  - $< (n-1)^{th} philosopher >< n^{th} philosopher >< 1^{st} philosopher >$

  Current philosopher at the middle, previous at left and next at right.

While developing seven main tasks, you are going to be in need of two helper functions. As you will guess, you are responsible for implementing these helper functions as well. **You do not have to implement and use these functions. However, if you need these features, then implementing them in their own function will make your code cleaner.**

- **int get_length(Node *list)** gets a linked list and returns it's size.
- **Philosopher *get_philosopher(Node *philosophers, int index)** gets linked list of the philosophers and an index. Returns philosopher pointer at given index.

4 Example

The main function will be as follows:

```
int main()
{
    Node *meals = NULL;
    Node *philosophers = NULL;
    Node *table = NULL;

    add_meal(&meals, "Kung Pao", 3);
    add_meal(&meals, "Tofu", 1);
    add_meal(&meals, "Wonton", 2);
    add_meal(&meals, "Fried Rice", 3);

    add_philosopher(&philosophers, "Lao Tzu", "Wonton", 1424);
    add_philosopher(&philosophers, "Confucius", "Tofu", 1145);
    add_philosopher(&philosophers, "Mozi", "Fried Rice", 1976);
    add_philosopher(&philosophers, "Shang Yang", "Fried Rice", 1555);

    print_list(meals, &print_meal_node);
    printf("--------------------\n");
    print_list(philosophers, &print_philosopher_node);

    place_philosophers(&table, philosophers);
    printf("--------------------\n");
    print_table(table);

    serve_meals(table, meals);
    printf("--------------------\n");
    print_list(meals, &print_meal_node);

    remove_philosopher(&table, 2, 4);
    printf("--------------------\n");
    print_table(table);
    return 0;
}
```

- Two singly linked lists and one table (circular singly linked list) will be created as NULL.
- Couple of meals and philosophers will be appended to the lists. We pass reference of the linked lists to the add functions to not lose original pointers.
- We will print meals and philosophers. As you can see, these print_list functions gets reference of the helper print functions as parameter.
- Philosophers will be placed around the table. Again, we pass table with its reference.
- We print the table with its idiosyncratic printing format.
- We serve meals and print meals list again to check new number of meals.
- We remove a philosopher from the table and print the table.

5 Output of the Example

```
Name: Kung Pao, count: 3
Name: Tofu, count: 1
Name: Wonton, count: 2
Name: Fried Rice, count: 3
--------------------
Name: Lao Tzu, favorite meal: Wonton, age: 1424
Name: Confucius, favorite meal: Tofu, age: 1145
Name: Mozi, favorite meal: Fried Rice, age: 1976
Name: Shang Yang, favorite meal: Fried Rice, age: 1555
--------------------
Mozi -> Confucius -> Lao Tzu
Confucius -> Lao Tzu -> Shang Yang
Lao Tzu -> Shang Yang -> Mozi
Shang Yang -> Mozi -> Confucius
--------------------
Name: Kung Pao, count: 3
Name: Tofu, count: 0
Name: Wonton, count: 1
Name: Fried Rice, count: 1
--------------------
Mozi -> Confucius -> Lao Tzu
Confucius -> Lao Tzu -> Mozi
Lao Tzu -> Mozi -> Confucius
```

6 Representation of the Scenario

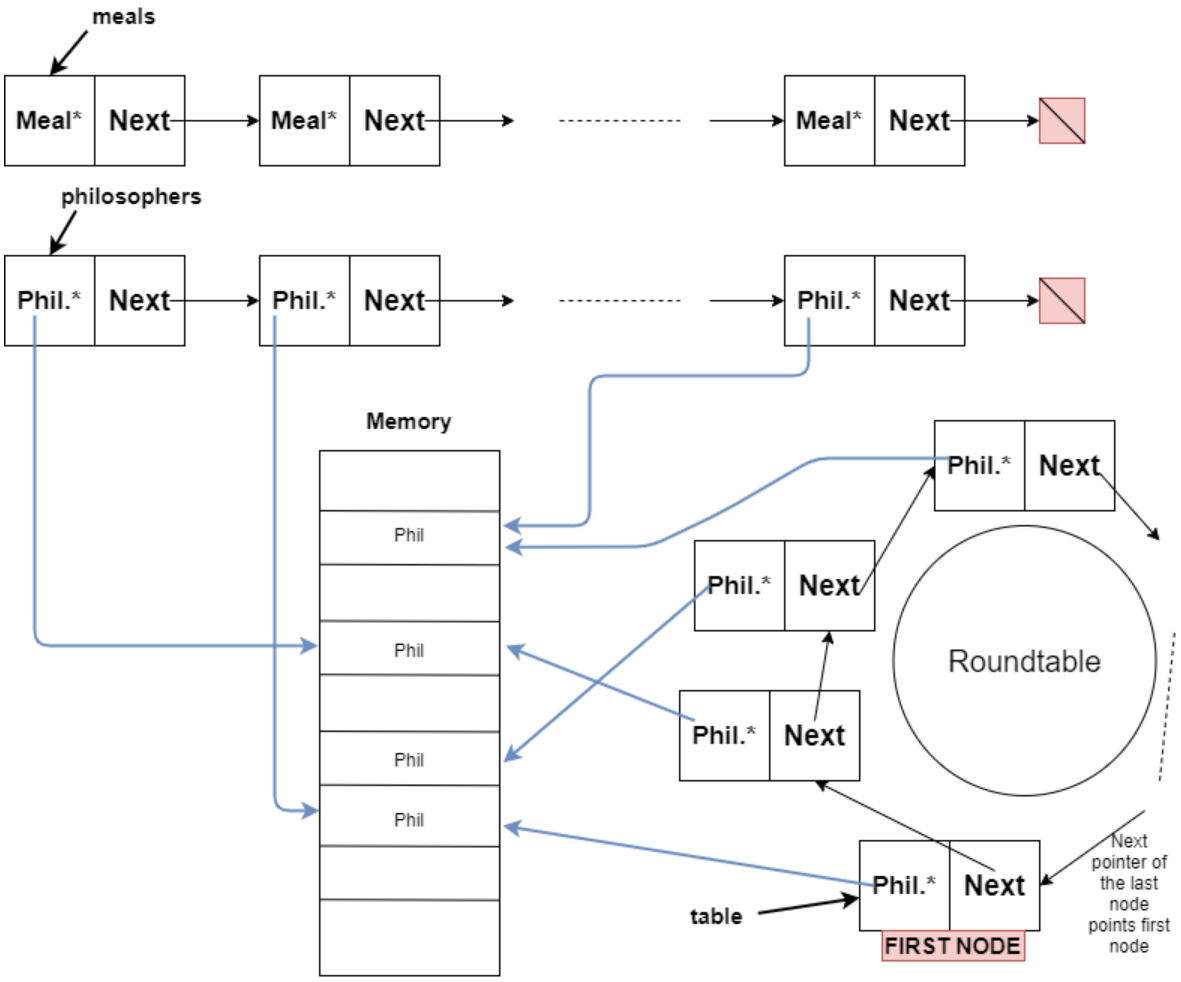

Figure 2: In this figure, black arrows represent "Node* next" members of nodes. Blue arrows represent philosopher pointers. As you can see, there are only one allocation on memory for each philosopher. The philosopher pointers in the original linked list and the circular linked list point to the same address allocated for that philosopher in memory.

7 Specifications

- Ages of the philosophers will be any positive integer less than 9999. This value is defined in *the3.h* header file as:

  ```
  MAX_VALUE = 9999
  ```

- Philosophers will not be agemate.
- There will be at least 3 philosophers around the table at anytime. (Even after removes)
- You can use **strcmp** function to compare philosophers' favorite meals with meals in the list. (You can only use strcmp from string.h)
- Philosophers' favorite meals will not be out of meals list and the meals will not reduce under 0.
- remove_philosopher function will not get any non-valid index.
- In print functions, there will be an extra newline character at the end. So, do not try to handle last item specifically to avoid extra newline character.

8 Regulations

- **Programming Language:** C
- **Libraries and Language Elements:**
  You should not use any library other than *"stdio.h"*, *"stdlib.h"* and *"string.h"*. You can use conditional clauses (switch/if/else if/else), loops (for/while), allocation methods (malloc, calloc, realloc). **You can NOT use any further elements beyond that (this is for students who repeat the course).** You can define your own helper functions.
- **Compiling and running:**
  DO NOT FORGET! YOU WILL USE ANSI-C STANDARTS. You should be able to compile your codes and run your program with given **Makefile**:

  ```
  >_ make the3
  >_ ./the3
  ```

- **Submission:**
  You will use CengClass system for the homework just like Lab Exams. You can use the system as an editor or work locally and upload the source files. Late submission IS NOT allowed, it is not possible to extend the deadline and **please do not ask for any deadline extensions**.
- **Evaluation:** Your codes will be evaluated based on several input files including, but not limited to the test cases given to you as an example. You can check your grade with sample test cases via CengClass system but do not forget it is not your final grade. Your output must give the exact output of the expected outputs. It is your responsibility to check the correctness of the output with the invisible characters. Otherwise, you can not get a grade from that case. If your program gives correct outputs for all cases, you will get 100 points.
- **Cheating: We have zero-tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations and will get 0. Sharing code between each other or using third party code is strictly forbidden. Even if you take a "part" of the code from somewhere/somebody else - this is also cheating. Please be aware that there are "very advanced tools" that detect if two codes are similar. So please do not think you can get away with by changing a code obtained from another source.