

Assignment 2: Pointers and Memory

Due Oct 17, 2018 by 4pm

Points 10

Available after Sep 27, 2018 at 12am


Assignment 2: A Help Centre Queueing System

Due: October 17, 2018 4 p.m.

Introduction

The Department of Computer Science has an undergraduate help centre where students can ask questions about their course work. Students swipe their T-cards to enter a software queue and wait their turn to see a TA. In this assignment, you will build an interactive system that could manage a similar help centre. Your program will **not** provide identical functionality to the current (or past) DCS help centre systems. You need to use **this handout** as the specification for your assignment (not the behaviour of the working system in DCS.)

Data structures

Most of your task is to complete the API for the data structures which are the basis for the help centre queue system. The header file `hcq.h`, provided in the starter code, defines the data structures you must use to build your program. Spend some time examining the code and this figure [hcqfig.pdf](#)  and make sure that you understand the structure you are trying to build before you begin to code. Ask questions at office hours or in the actual help centre or on Piazza.

The `hcq` data structures include two linked structures (one for students and one for TAs) and one array (for courses.)

The array of courses has elements of type `struct course` and is created at the start of your program using initial values read from a configuration file. Each course struct has information about the course (not all of it shown in the figure) and pointers that reference the first and last students waiting in the queue for this course.

The TA list is a singly-linked list of `struct ta` elements kept in reverse order on arrival. The TA who arrived most-recently is at the head of the list. Each `ta` struct contains a pointer to the `struct student` representing the student who is currently being helped by this TA. These students are **no longer** in any queue. Notice that the middle TA in the list does not currently have a student.

The data structure of `struct student` objects that represents students waiting for help contains two linked-lists. Each student has a pointer to the `next_overall` student and the `next_course` student. Consider student `s` who asked a question about course `c`. At any point in time, `s`'s `next_overall` pointer, points to the student remaining in the queue who arrived most recently after `s`. `s`'s `next_course` pointer, points to the

next student still in the queue who has a question about course `c`. Either of these pointers can be null if there is no student after `s` or no student after `s` from the same course.

One important thing to notice about a student, is the `arrival_time` field of the struct. For students currently waiting in the queue, this represents the time that they arrived in the queue to ask a question. For students who are being helped by a TA, this field should hold the time that the TA started to help the student. You will need these values in order to calculate the total amount of time students from a particular course spent waiting to be helped and also the total amount of time TAs spent helping students from a particular course. Your textbook has a nice discussion of how time is represented in C that will help you understand the type `time_t`.

The variables shown in the figure in orange are the statically-allocated pointers to these data structures. They are declared in the main function of `helpcentre.c`. Spend some time exploring the starter code, looking at the figure and rereading this description until you are certain that you understand the data structures you need to create and maintain.

Queue Operations

The program itself is divided into two main components: a basic hcq API that provides functions to manipulate the hcq data structures and a user-interface that parses command-line instructions and calls the appropriate functions from the API. We are providing the user-interface code in the file `helpcentre.c`, which you are not allowed to change. Read it carefully to understand how it works.

Your help centre queue will be manipulated through a boring (but simple) command line interface that supports the following operations. These operations are not fully specified here. You will need to read the starter code including comments to see the full required behaviour.

Command	Description
<code>stats_by_course course</code>	Prints the current stats for a course: how many students are waiting, how many are currently being served, how many have already served, how many students were waiting for this course and gave up, and the total time that has been spent waiting for this course and answering questions for this course. Much of this function is provided so that the formatting of the output is consistent. Do not change the provided code; Only add to it.
<code>add_student name course</code>	Adds a new student by this name to the data structure and records that the student has a question for this course.
<code>add_ta name</code>	Adds a TA by this name.
<code>remove_ta name</code>	Removes the TA by this name.
<code>print_all_queues</code>	Prints the full list of courses. For each course, prints the total number and names (in order) of students queued up for that course. We have provided print statements so that you can get the formatting correct for testing. You can change the variable names in the print statements, but don't change the format strings. Students currently being served by a TA are no longer in a queue and so are not shown.

give_up student_name	Removes the student from the waiting queue and adjusts any stats as needed because this student is giving up before they are called for their turn. The time the student spent waiting <i>before giving up</i> is including in the total waiting time for this course.
print_currently_serving	Prints the complete list of TAs. For each TA, prints the student being helped and for which course.
next TA_name	The TA is assigned to the next student in the overall queue. If the TA was helping a student when this command is given, it means they are finished with that student. If there is no student waiting, it isn't an error but simply means that the TA no longer has a current student.
next TA_name [Course]	If the next command has two arguments, then the second is a course code. In this case, the TA gets the next student from this particular course. If the course has nobody waiting or the course number is invalid, the TA does not take a student from another course but instead has no current student.
print_full_queue	This function prints a list of each student in the entire queue in overall order. Each student is shown with their name and course. This function will not be tested for A2, so you do not need to implement it. However, you might find that it is helpful for testing and debugging your solution to other functions. You might also find it helpful to print the arrival time of each waiting student.
help	Lists the commands available.
sleep	Pauses the program for 2 seconds. Useful for testing with batch file.
quit	Closes the helpcentre program.

The helpcentre executable can be started from the command line with either one or two arguments as follows:

```
./helpcentre config_file [filename]
```

If helpcentre is run with only one argument (the name of the configuration file), it starts in interactive mode where it displays a prompt and waits for one of the above commands from standard input. When run with two arguments, helpcentre expects the second argument to be the name of a file that contains one helpcentre command per line. The commands are those from the list above. The program will execute those commands from the file. When run with commands from a file, execution happens quickly - so quickly that the waiting times and service times are almost all zero. To simulate time passing, we have included a sleep() command in the API.

Starter code

Log into MarkUs and navigate to the A2: Pointers and Memory. In the same way as for A1, this triggers the starter code for this assignment to be committed to your repository. You will be working alone on this

assignment, and the URL for your Git repository can be found on the right hand side of the page. Pull your git repository. There should be a new directory named A2. All of your code for this assignment should be located in this directory.

This directory contains a Makefile and three files called `helpcentre.c`, `hcq.h`, and `hcq.c` that provide the implementation of a skeleton of the help centre data structures. `helpcentre.c` provides the code to read and parse the user commands described above, and then calls the corresponding C function for each user command. Read the starter code carefully to familiarize yourself with how it works.

Function prototypes for all functions implementing the various user commands are provided in `hcq.h`. So that the starter code compiles without errors, these functions are currently implemented in `hcq.c` as stubs (i.e. functions whose body is empty except for a return statement). In addition, there are prototypes and stub implementations for some helper functions that you will use to implement the user-command functions. Your task will be to complete the implementation of all these functions. You are encouraged to create additional helper functions to avoid duplicate code.

We have also provided a sample configuration file `courses.config` and a sample input file `batch_commands.txt` that contains a series of commands that will execute in batch mode.

Your tasks

You will complete the functions defined in `hcq.c`. The comment above each function stub further explains the behaviour of the function.

Note that you are not allowed to modify `helpcentre.c` or `hcq.h`. The only C file you are modifying is `hcq.c`. You may add new functions to `hcq.c`, but do not change the names or prototypes of the functions we have given you. To add helper functions to `hcq.c` you should not be adding their declarations into `hcq.h` which you are not allowed to change. Either add the declarations at the top of `hcq.c` or define the helper functions before you call them.

Part I: Configuring the Array of Courses

Start by implementing and testing the function `config_course_list` which sets up the array of courses. The format of the configuration file is as follows:

1. The first line is a single integer and is the number of courses.
2. Each subsequent line represents a course.
3. Each course line begins with a 6 character `code` followed by a single space. The rest of the line is the course description. The entire line (including newline characters and null terminators) will fit into a buffer of length `INPUT_BUFFER_SIZE`.

Complete the function `find_course` and now you can complete and call `stats_by_course` to partially test your work so far. Of course the stats won't be very interesting since you haven't added any students yet.

Once you have these three functions completed, you should be able to run your helpcentre executable from the commandline and call `stats_by_course`, `help`, and `quit`.

Part II: Adding students and TAs and giving up

Next, write the code for the functions that are called for the user commands `add_student` and `remove_ta` and `print_all_queues`. Once you've done that, complete the functions needed when a student gives up waiting and then another one of the printing functions so that you can see the effects of working functions. We recommend you implement functions in this order:

- `Student *find_student(Student *stu_list, char *student_name)`
- `Course *find_course(Course *courses, int num_courses, char *course_code)`
- `int add_student(Student **stu_list_ptr, char *student_name, char *course_num, Course *courses, int num_courses)`
- `void release_current_student(Ta *ta)`
- `int give_up_waiting(Student **stu_list_ptr, char *student_name)`
- `void print_all_queues(Student *stu_list, Course *courses, int num_courses)`

At this point, you may want to implement `print_full_queue`. Although it will not be marked (and so you don't have to carefully match an output specification), it might help with the testing and debugging of your other functions.

Part III: Calling students for help

Next implement the functions needed for the two user commands that let TAs call a student to begin helping them: `next_student` and `next_student_by_course`. Notice that the API functions have very similar descriptions which suggests that they probably have some common functionality that may motivate the use of a helper function.

For this part you should also implement the function `print_currently_serving` perhaps doing this first in this section in order to have it available to help debug your other functions.

The API functions you need here are:

- `void print_currently_serving(Ta *ta_list)`
- `TA *find_ta(Ta *ta_list, char *ta_name)`
- `int take_next_overall(char *ta_name, Ta **ta_list_ptr, Student **stu_list_ptr)`
- `int take_next_course(char *ta_name, Ta **ta_list_ptr, Student **stu_list_ptr, char *course_code, Course *courses, int num_courses)`

Carefully read the docstrings on these functions to understand how they behave.

Part IV: Checking your statistics


At the time that you implemented `stats_by_course`, you didn't have any students or TAs in the system to check your numbers. You may find that you need to go back and add code to your already-implemented functions to correctly compute the required statistics. Pay particular attention to the fact that when a TA leaves, the current student should be counted as served and the time spent serving that student should contribute to the total helping time for the appropriate course. When a student gives up, the time they spent waiting for service (even though they were never served) should be included in the total waiting time for the course.

Error handling

The comments for the functions you are to implement define nearly all the error conditions you might encounter and tell you how to handle them. The main additional expectation is that you check the return value from `malloc` and terminate the program if `malloc` fails.

You should assume that the configuration file is correctly formatted.

What to hand in

Commit to your repository in the A2 directory your revised `hcq.c` file. Since you are not allowed to change `hcq.h`, `helpcentre.c`, and the provided Makefile, we will use the original versions for compiling and testing your code. **The markers must be able to run  such that helpcentre is compiled with no warnings.**

Coding style and comments are just as important in CSC209 as they were in previous courses. Use good variable names, appropriate functions, descriptive comments, and blank lines. Remember that someone needs to read your code. You should write extra helper functions.

Please remember that if you submit code that does not compile, it will receive a grade of 0. The best way to avoid this potential problem is to write your code incrementally. The starter code compiles without warnings and runs. As you solve each small piece of the problem make sure that your new version continues to compile without warnings and runs. Get a small piece working, commit it, and then move on to the next piece. This is a much better approach than writing a whole bunch of code and then spending a lot of time debugging it step by step.