CW-B1 Audioscrobbler Plays

Released: Nov 9th 2016
Deadline: Nov 18th 2016
Weight: 15 %

This coursework carries 15 % towards your final mark. Once you have completed the two tasks, submit one C source `query_plays.c`. by clicking the "Submit CW-B1" link on Moodle. You can ask a lab assistant to give you a mark and feedback during the lab hours, or in one of the coursework marking sessions which will be announced on Moodle. The marking scheme is as follows:

- Task 1) 1 mark each for (a)-(c).
- Task 2)
  - 2 marks for each of (a), (c), (d), and (e)
  - 1 mark for (b)
  - 3 marks for (f)
  - (g) will be marked later

Introduction

Audioscrobbler was an online service for storing music listening patterns. A data set was published to the public domain containing a list of artists and user plays.

We will only use the data in the file `user_artist_data.txt`, which we will refer to as the "play" data set. Here are the first three lines of this file:

```
1000002 1 55
1000002 1000006 33
1000002 1000007 8
```

Each line has three columns, a user id, an artist id, and a play count. A line gives the number of times a user has played a given artist. For example, the first line of the file says that the user with user id 1000002 has played the artist with artist id 1, 55 times.

We will store the play data in a linked list using the following struct.

```
struct play
{
  int user_id;
  int artist_id;
```

```
  int playcount;
  struct play *next;
};
```

Before starting the coursework, you need to download the play data set from Moodle. You should also download the C source file `query_plays_start.c` which contains some initial definitions.

Task 1 - Structs and Malloc

In this task, we will create functions for printing, and allocating and freeing memory for individual `play` structs.

a) Create a function with the following type declaration

```
void print_play(struct play *p);
```

which outputs the user id, the artist id, and the play count of the play pointed to by the variable `p` on the format of this example.

```
user: 100 artist: 200 count: 5
```

If `p` is a `NULL` pointer, then the function should instead print `NULL`.

b) Implement a function with the following type declaration which allocates memory on the heap for one `play` struct, and which fills the struct with the values `user_id, artist_id`, and `count` which are provided as arguments to the function. The function should return a pointer to the new struct, or `NULL` if there was a memory allocation error.

```
struct play *create_play(int user_id, int artist_id, int count);
```

c) Define a function with the following type declaration which deallocates (frees) the memory used for the struct `p`.

```
void free_play(struct play *p);
```

If you have completed this task correctly, then the code

```
struct play *test_play = NULL;
print_play(test_play);
test_play = create_play(1,2,3);
print_play(test_play);
free_play(test_play);
```

should print the following lines

```
NULL
user: 1 artist: 2 count: 3
```

Task 2 - Linked Lists

In this task, we will create a linked list to store the elements in the user play database.

(a) Implement a function `add_play` with the type declaration below which takes two arguments, `head` which is a pointer to a linked list of `play` structs, and `newp` which is a single struct which should be added to the *beginning* of the linked list. If `newp` is a `NULL` pointer, then nothing should be added to the linked list. The function should return the head of the new linked list.

```
struct play *add_play(struct play *head, struct play *newp);
```

(b) Create a function `print_plays` which prints all plays in a linked list using the `print_play` function from Task 1.

```
void print_plays(struct play *head);
```

(c) We would like to count the number of plays in the play database. Construct a function `count_plays` which given a linked list of `play` structs, counts the number of plays. Note that we are not looking after the number of `play` structs in the linked list, but the sum of the play count values stored in those structs.

```
int count_plays(struct play *head);
```

If you have completed Task 2 (a), (b), and (c), correctly, then the following test code

```
struct play *a = create_play(1,2,3);
struct play *b = create_play(4,5,6);
a = add_play(a, NULL);
a = add_play(a, b);
print_plays(a);
printf("There are %d plays in a.\n", count_plays(a));
```

should produce the output

```
user: 4 artist: 5 count: 6
user: 1 artist: 2 count: 3
There are 9 plays in a.
```

(d) Create a function which reads all plays from the play data file into a linked list of `play` structs. The function should return `NULL` if an error occurred, otherwise it should return a pointer to the beginning of the linked list. Hint: Use the `create_play` and `add_play` functions.

```
struct play *read_plays(char *file_name);
```

(e) Create a function `free_plays` which deallocates all the elements in a linked list of plays using the `free_play` function from Task 1. The function should have the following type declaration.

```
void free_plays(struct play *head);
```

(f) We would like to learn more about particular users. Implement a function `filter_user` which removes all plays from a linked list, except those belonging to a particular user. Remember to free memory when removing elements from the struct. The function should be declared as follows.

```
struct play *filter_user(int user_id, struct play *head);
```

Hint: Create a new empty linked list, and add the relevant plays to this list using the `add_play` function.

(g) The final task in this coursework is to integrate all the functionality we have implemented into a program `query_plays` which takes two or three arguments. The first argument is the name of the play data file, and the remaining arguments are one of the following commands:

- ▷ `c <userid>` counts all the plays in the data base, or if the optional argument `<userid>` is provided, only the plays corresponding to this user.
- ▷ `p <userid>` prints all the plays in the data base, or if the optional argument `<userid>` is provided, only the plays corresponding to this user.

The program should print an appropriate error message, free allocated resources (memory and file), and halt if one of the following error occurs: memory allocation error, file error, and incorrect number of arguments provided.

Note: You should use the functions from the previous tasks, including `filter_user`, to solve Task 2 (g).

Example:

```
[pkl@gamma cw-b1]$ ./query_plays user_artist_data.txt c
371638969
[pkl@gamma cw-b1]$ ./query_plays user_artist_data.txt c 2428613
1581
[pkl@gamma cw-b1]$ ./query_plays user_artist_data.txt p 2428613
user: 2428613 artist: 1000024 count: 1
user: 2428613 artist: 1000033 count: 4
user: 2428613 artist: 1000107 count: 14
...
```

The last program invocation only shows the first three lines of the output.