Sharath Mahadevan

Assignment 2 Questions

1. In this assignment I asked you provide an implementation for the get\_student(...) function because I think it improves the overall design of the database application. After you implemented your solution do you agree that externalizing get\_student(...) into it's own function is a good design strategy? Briefly describe why or why not.

**ANSWER**: Yes, I do believe that externalizing get\_student() into it's own function is a good design strategy. This allowed us to centralize the logic for retrieving a student. What this does is make our code cleaner, and we follow better design principles by doing this. It also makes it easier to access and reuse this code if/when needed across the project.

2. Another interesting aspect of the get\_student(...) function is how its function prototype requires the caller to provide the storage for the student\_t structure:

int get\_student(int fd, int id, student\_t \*s);

Notice that the last parameter is a pointer to storage provided by the caller to be used by this function to populate information about the desired student that is queried from the database file. This is a common convention (called pass-by-reference) in the C programming language.

In other programming languages an approach like the one shown below would be more idiomatic for creating a function like get\_student() (specifically the storage is provided by the get\_student(...) function itself):

//Lookup student from the database

// IF FOUND: return pointer to student data

// IF NOT FOUND: return NULL

student\_t \*get\_student(int fd, int id){

student\_t student;

bool student\_found = false;

//code that looks for the student and if

//found populates the student structure

//The found\_student variable will be set

//to true if the student is in the database

//or false otherwise.

if (student\_found)

return &student;

else

return NULL;

}

Can you think of any reason why the above implementation would be a very bad idea using the C programming language? Specifically, address why the above code introduces a subtle bug that could be hard to identify at runtime?

**ANSWER**: This implementation would be a bad idea within the C programming language because returning a pointer to a local variable(such as student) will not work. This variable will be allocated on the stack. Once this function returns, the pointer will become invalid, and when trying to access this pointer, an error will be presented such as undefined behavior.

3. Another way the get\_student(...) function could be implemented is as follows:

//Lookup student from the database

// IF FOUND: return pointer to student data

// IF NOT FOUND or memory allocation error: return NULL

student\_t \*get\_student(int fd, int id){

student\_t \*pstudent;

bool student\_found = false;

pstudent = malloc(sizeof(student\_t));

if (pstudent == NULL)

return NULL;

//code that looks for the student and if

//found populates the student structure

//The found\_student variable will be set

//to true if the student is in the database

//or false otherwise.

if (student\_found){

return pstudent;

}

else {

free(pstudent);

return NULL;

}

}

In this implementation the storage for the student record is allocated on the heap using malloc() and passed back to the caller when the function returns. What do you think about this alternative implementation of get\_student(...)? Address in your answer why it work work, but also think about any potential problems it could cause.

**ANSWER**: This implementation works by allocating memory for the heap. It seems like this implementation will work, however I believe that this could lead to memory leaks if the caller forgets to free the memory allocated to the heap. This would require some sort of dynamic memory allocation which could be inefficient.

4. Lets take a look at how storage is managed for our simple database. Recall that all student records are stored on disk using the layout of the student\_t structure (which has a size of 64 bytes). Lets start with a fresh database by deleting the student.db file using the command rm ./student.db. Now that we have an empty database lets add a few students and see what is happening under the covers. Consider the following sequence of commands:

> ./sdbsc -a 1 john doe 345

> ls -l ./student.db

-rw-r----- 1 bsm23 bsm23 128 Jan 17 10:01 ./student.db

> du -h ./student.db

4.0K ./student.db

> ./sdbsc -a 3 jane doe 390

> ls -l ./student.db

-rw-r----- 1 bsm23 bsm23 256 Jan 17 10:02 ./student.db

> du -h ./student.db

4.0K ./student.db

> ./sdbsc -a 63 jim doe 285

> du -h ./student.db

4.0K ./student.db

> ./sdbsc -a 64 janet doe 310

> du -h ./student.db

8.0K ./student.db

> ls -l ./student.db

-rw-r----- 1 bsm23 bsm23 4160 Jan 17 10:03 ./student.db

For this question I am asking you to perform some online research to investigate why there is a difference between the size of the file reported by the ls command and the actual storage used on the disk reported by the du command. Understanding why this happens by design is important since all good systems programmers need to understand things like how linux creates sparse files, and how linux physically stores data on disk using fixed block sizes. Some good google searches to get you started: "lseek syscall holes and sparse files", and "linux file system blocks". After you do some research please answer the following:

Please explain why the file size reported by the ls command was 128 bytes after adding student with ID=1, 256 after adding student with ID=3, and 4160 after adding the student with ID=64?

**ANSWER:** The ls command shows the size of the file. As records are added to this file, the size of the file increases. Each student record is 64 bytes so adding in students with ids 1, 3 and 64 would result in the file size: 128 -> 256 -> 4160 bytes

Why did the total storage used on the disk remain unchanged when we added the student with ID=1, ID=3, and ID=63, but increased from 4K to 8K when we added the student with ID=64?

**ANSWER:** The "du" command shows the disk space used for a file. When adding in data to a file, there are holes where no data is stored. When adding in the student\_id 64, a block is filled with that data, which causes increased disk usage. This is what causes the disk usage to increase from 4k to 8k.

Now lets add one more student with a large student ID number and see what happens:

> ./sdbsc -a 99999 big dude 205

> ls -l ./student.db

-rw-r----- 1 bsm23 bsm23 6400000 Jan 17 10:28 ./student.db

> du -h ./student.db

12K ./student.db

We see from above adding a student with a very large student ID (ID=99999) increased the file size to 6400000 as shown by ls but the raw storage only increased to 12K as reported by du. Can provide some insight into why this happened?

**ANSWER:** Adding in a student with a large ID such as 99999 creates a gap within the file, which increases the file size to 6400000 bytes. However the actual disk usage remains low because the file system handles the sparse files, and only allocates space for the data and not the gaps within the file.