1. Introduction

1.1. **Goal**

For this assignment, you will write a multi-process program in C, in the Ubuntu Linux environment of the course VM, that manages and regulates access to the books in a library.
Your program will consist of two processes: (1) a server process that regulates access to the book data of the library and fulfills requests from the client process on demand; and (2) a client process that interacts with the end user and enables them to view the books of the library, and allows them to check books in and out of the library.

The two processes will communicate with each other using TCP/IP sockets.

1.2. **Learning Outcomes**

With this assignment, you will:

- write a program in C that is distributed over two processes that communicate using TCP/IP sockets
- practice implementing a signal handler
- read data into program memory using file I/O in C

1.3. **Required background**

To begin this assignment, you must understand the following concepts from the course material:

- section 5.1 - 5.3: distributed systems, process management, signals, sockets
- section 6: file I/O

If you are not familiar with this material, you **must** review it before you can start this assignment.

1.4. **Expectations**

In accordance with the course outline, paragraph 8.5:

- **only *working code* will earn marks**; code that does not compile in the course VM will **not** earn marks; code that cannot be demonstrated to actually work, because it does not execute in the VM, or is not called, or is commented out, or does not print data, will earn no more than 50% of the maximum marks for each marking component that cannot be demonstrated to work
- **only code that is implemented *in accordance with the instructions and constraints* stated in this document will earn marks**; code that does not follow the instructions and constraints, and code that uses techniques and/or libraries and/or functions that are not explicitly supported in this course, will **not** earn marks, even if the code is working

Students are expected to seek all required clarifications from the instructor in the appropriate forum in *Brightspace*.

2. Instructions

The server process begins by initializing the book data in a library. The data is contained in a plain text file that is provided as part of the a4-posted.tar file posted in *Brightspace*. Once the library data is loaded into the server process memory, the server waits for a connection request to come in from a client process. Once a client process connects to the server, the server fulfills requests from that client until the client closes the connection. When that happens, both the server and client processes terminate.

The client process provides a user interface for the end user, in the form of a main menu. The client has *no library data* loaded into it, and it has no knowledge of book structures. If the user chooses to view the books in the library, the client process must request this information from the server, then print to the screen the string-formatted data received by the server. If the user chooses to check a book in or out of the library, the client forwards this request to the server, and the server performs the required work.

2.1. **Understand the base code**

You will begin by understanding the base code provided in the a4-posted.tar file in *Brightspace*.

2.1.1. The defs.h header file contains all the structure data types that you must use in this program. You must update this file with all the function prototypes as forward references.

2.1.2. The connect.c source file contains all the basic TCP/IP socket communication functions that you need for this program. You will recognize the code that we worked on in the coding examples of section 5.3, program #4, but this code is now organized for you in modular functions.

2.1.3. The server.c source file contains a skeleton main() function for the server process, as well as a global variable to store the server's listening socket:

(a) yes, we have done the unforgivable here: we are using a global variable to store the listening socket; this is a regrettable design choice, but it's a necessity in this case

(b) this is the **ONLY** global variable that is permitted in this program

2.1.4. The client.c source file contains a skeleton main() function for the client process, as well as a clientMenu() function that presents a menu to the client's end user and reads their selection.

2.1.5. The booksData.txt file contains the data that must be read into the server process's memory.

2.1.6. The provided Makefile creates both executables required for this program, using the source file names described in instruction 2.2.

2.2. **Design the separation of functions into files**

As you begin implementing the functions described in instruction 2.3 and beyond, you are required to separate these functions into at least five (5) source files for this assignment. Your program must be organized as follows:

2.2.1. server.c contains the main() function for the server process, the helper functions that assist in managing the server control flow and cleanup, and a signal handler function

2.2.2. lib.c contains the library initialization and manipulation functions

2.2.3. books.c contains the functions that manipulate the book structures and collection

2.2.4. client.c contains the main() function for the client process, as well as the client menu function

2.2.5. the provided connect.c file contains the functions that handle TCP/IP socket communications be-tween the server and client processes

2.3. **Implement the book manipulation functions**

Your program requires several functions to process the book data and the associated collection.

2.3.1. Implement the void initList(BookListType *list) function that initializes the required fields of the given book list.

NOTE: The singly linked list in this program must be implemented exactly as we did in the course material, section 3.3, program #3, except that the list in this assignment must keep track of the list's tail, as well as its head.

2.3.2. Implement the void initBook(int i, char *t, char *a, int y, BookStatusType st, BookType **book) function that does the following:

(a) dynamically allocate the memory for a new BookType structure

(b) initialize the new book's id, title, author, year, and availability status from the parameters

(c) use the book parameter to "return" the new book structure

NOTE: You may need to refresh your memory on how data is "returned" using parameters, which we covered in the course material of section 3.2, program #4.

2.3.3. Implement the void addBook(BookListType *list, BookType * b) function that inserts the new book into the book collection, **directly in its correct position**, in ascending order by title. Do not add to the end of the list and sort. Make sure that the list head and tail are both updated, if required.

2.3.4. Implement the int findBook(BookListType *list, int id, BookType **b) function that tra-verses the given book collection to find the book with the given id, and returns the found book pointer using the b parameter. This function returns a success flag if the book is found, and an error flag otherwise.

2.3.5. Implement the void formatBook(BookType *b, char *outStr) function that formats all the in-formation for a single book into one long string, and stores that string in the outStr parameter, which is “returned” to the calling function. Specifically:

(a) all the book’s data must be formatted into a single long string: book id, title, author, year, and current status

(b) the formatted data must be stored in the string as space-separated, fixed-width fields, for ease of printing in a later step; hint: the string library’s sprintf() function is helpful here

(c) all data must be formatted in a way that is meaningful to the user; this means that you must implement a helper function to convert the current status enumerated data type to a string value

2.3.6. Implement the void formatBooks(BookListType *list, char *outStr) function that traverses the book collection list, formats the information for each individual book, and populates the outStr parameter with the entire contents of the list, with each book separated from the other with a “new line” character. Specifically:

(a) your code must loop through all the books in the list

(b) for each book, you must:

  (i) call an existing function to format that single book’s data into a temporary string

  (ii) append that temporary string to the existing information in the outStr parameter

(c) once you have formatted all the books into the very long outStr string, you must append to it the formatted book data found at the head and at the tail of the list; you must refer to the sample execution in the introduction video for an example of what is required here

2.3.7. Implement the void cleanupList(BookListType *list) function that deallocates all the mem-ory that was dynamically allocated for the book collection.

2.4. **Implement the library manipulation functions**

You will implement the functions that initialize all the data in the library by loading the book information from the provided data file into the server process’s memory.

2.4.1. Implement the void loadBooks(BookListType *books) function that reads in all the data from the provided booksData.txt file, creates and initializes a BookType structure for each book in the data file, and adds each book to the given book collection. This function must automatically generate a unique identifier for each new book created.

2.4.2. Implement the void initLibrary(LibraryType **library, char *n) function that does the fol-lowing:

(a) dynamically allocate the memory for a new LibraryType structure, and initialize the new library’s name to the provided parameter

(b) initialize the new library’s book collection

(c) load the book data from the data file, and store it into the library’s book collection

(d) print the contents of the book collection to the screen by formatting it into a long string, and printing the resulting string to the screen

(e) you must reuse functions that you implemented in a previous step everywhere possible

(f) use the library parameter to “return” the new library structure

2.4.3. Implement the int checkOutBook(LibraryType *lib, int bookId) function that does the fol-lowing:

(a) find the book with the given id in the library’s book collection

(b) check that the book is currently checked in (we cannot check out a book that is not checked in)

(c) if an error occurs, return the corresponding error flag: C_BOOK_NOT_FOUND, or C_BAD_STATUS

(d) if no error occurs, set the book’s availability status to checked out, and return a success flag

2.4.4. Implement the int checkInBook(LibraryType *lib, int bookId) function that as follows:

(a) find the book with the given id in the library’s book collection

(b) check that the book is currently checked out (we cannot check in a book that is not checked out)

(c) if an error occurs, return the corresponding error flag: C_BOOK_NOT_FOUND, or C_BAD_STATUS

(d) if no error occurs, set the book’s availability status to checked in, and return a success flag

2.4.5. Implement the void cleanupLibrary(LibraryType *lib) function that uses existing functions to cleanup the book collection in the given library, and deallocates the memory for the library itself.

2.5. **Implement the server functionality**

In this section, you will implement the server process control flow.

NOTE #1: The control flow of your server process behaves as a loop. First, the server waits for a connection request to come in from a client process. Once a connection request comes in and the server is connected to that client, the loop executes and fulfills requests from that client process, until the client chooses to close the connection. When this happens, the server control flow cleans up its resources and terminates.

NOTE #2: The server process only exits the loop (and terminates completely) if it receives a “close” request from a client process, or if it receives a SIGUSR1 signal from the command line.

You will implement the server functionality as follows:

2.5.1. Implement the server process’s main() function:

(a) declare a library structure (as a LibraryType pointer) and a client socket as local variables
(b) initialize and data load the library using an existing function
(c) set up the server’s listening socket using a provided function
(d) install a signal handler to handle the SIGUSR1 signal (see the coding examples of section 5.3, program #1); the signal handler is the handleSig1() function, which is described in a later step
(e) wait for a connection request to come in from a client process, using the global listening socket and the local client socket
(f) once a connection is established, call a helper function (implemented in the next step) to serve all the requests on the client socket until the client closes the connection

2.5.2. Implement the void serveOneClient(int clientSocket, LibraryType *lib) function that serves all the requests from the client process connected at the given client socket, using the given library data.

This function body consists of a loop that does the following:

(a) wait to receive data from the client process on the client socket
(b) take the first character of the received data buffer, and convert it to a numeric value; this will indicate what action is requested by the client, and it should map to one of the values in the provided RequestType enumerated data type
(c) if the requested action is REQ_RET_BOOKS, your code must format the given library’s entire book collection into one long string, and send that string back to the client process
(d) if the requested action is REQ_CHECK_OUT, the client’s end user wants to check out a book:
    (i) using the C string library function sscanf(), parse the received data buffer into the two following fields: the requested action, and a book id
    (ii) call a helper function to check out of the library the book with the received book id, and store the helper function’s return code (its return value that indicates either success or a specific error)
    (iii) format that return code into a string, and send it to the client process to confirm success of the check out operation, or to indicate the specific error that occurred
(e) if the requested action is REQ_CHECK_IN, the client’s end user wants to check in a book:
    (i) using the C string library function sscanf(), parse the received data buffer into the two following fields: the requested action, and a book id
    (ii) call a helper function to check in to the library the book with the received book id, and store the helper function’s return code
    (iii) format that return code into a string, and send it to the client process to confirm success of the check in operation, or to indicate the specific error that occurred
(f) if the requested action is REQ_CLOSE, it means that the client’s end user has exited the main menu and wants the server process to terminate; your code must close the client socket, call a helper function to clean up all server resources, and exit the program using the exit() function from the stdlib library

NOTE #3: The control flow never explicitly breaks out of the above loop. Both server termination conditions listed in Note #2 above must result in a call to exit(). This is the only successful way that the server process ever terminates.

2.5.3. Implement the void closeAll(LibraryType *lib) function that cleans up all the server resources, in preparation for the process terminating. This includes closing the listening socket and cleaning up all the memory for the data in the given library.

2.5.4. Implement the void handleSig1(int) function that cleans up the server resources as much as possible. This means closing the listening socket and using the exit() function to terminate the process. Unfortunately, the signal handler is unable clean up the library data.

2.6. **Implement the client functionality**

You will implement the client process's main() function to display the main menu to the end user, and communicate with the server process in order to fulfill each user request.

It's important to note that the client process has no knowledge of any data structures at all. It does not create or access books, or the library.

The client process's main() function does the following:

2.6.1. declare a client socket and initialize it using a provided function

2.6.2. repeatedly display the main menu to the end user until they choose to exit

2.6.3. if the user chooses to print the library's books:

(a) formulate a request string containing the REQ_RET_BOOKS request type
(b) send the request string to the server process on the client socket
(c) wait to receive the response from the server; this will be a long string of formatted books
(d) print the received books as a single string to the screen

2.6.4. if the user chooses to check out a book:

(a) prompt the user to enter a book id
(b) formulate a request string containing the REQ_CHECK_OUT request type and the book id, all for-matted as space-separated values in the string
(c) send the request string to the server process on the client socket
(d) wait to receive the response from the server; this will be the return code from the server's check out operation
(e) print either a confirmation message that the operation was successful, or print a detailed error message corresponding to the return code

2.6.5. if the user chooses to check in a book:

(a) prompt the user to enter a book id
(b) formulate a request string containing the REQ_CHECK_IN request type and the book id, all format-ted as space-separated values in the string
(c) send the request string to the server process on the client socket
(d) wait to receive the response from the server; this will be the return code from the server's check in operation
(e) print either a confirmation message that the operation was successful, or print a detailed error message corresponding to the return code

2.6.6. if the user chooses to exit the main menu:

(a) formulate a request string containing the REQ_CLOSE request type
(b) send the request string to the server process on the client socket
(c) close the client socket, and terminate the client process

3. Constraints

Your program must comply with all the rules of correct software engineering that we have learned during the lectures, including but not restricted to:

3.1. The code must be written using the default C standard that is supported by the course VM, and it must compile and execute in that environment. It must not require the installation of libraries or packages or any software not already provided in the course VM.

3.2. Your program must be correctly designed, and it must be separated into modular, reusable functions.

3.3. Do not use any global variables, except where explicitly permitted in the instructions.

3.4. If base code is provided, do not make any unauthorized changes to it.

3.5. Your program must reuse the functions that you implemented, everywhere possible.

3.6. Your program must perform all basic error checking.

3.7. Do not duplicate any structure data. You may have multiple pointers to the same data, but not multiple copies of the same data.

3.8. Compound data types must always be passed by reference, never by value.

3.9. Return values must be used only to indicate function status (success or failure). Except where otherwise noted in the instructions, data must be returned using parameters, and never using the return value.

3.10. All dynamically allocated memory must be explicitly deallocated.

3.11. All functions and parameters must be documented, as indicated in the course material, section 1.2.

4. Submission

4.1. One tar or zip file that includes:

(a) all source and header files, as well as any required data files, if applicable

NOTE #1: Do not include object files, executables, hidden files, or duplicate files or directories in your submission.

NOTE #2: Your Makefile **must** separate the compiling of source code into object code, and the linking of object code into two executables.

5. Grading

5.1. **Marking components:**

- 25 marks: correct implementation of book manipulation functions
- 28 marks: correct implementation of library functionality functions
- 27 marks: correct implementation of server functionality functions
- 20 marks: correct implementation of client functionality functions

5.2. **Execution requirements:**

5.2.1. all marking components must be called and execute successfully in order to earn full marks

5.2.2. all data handled must be printed to the screen for marking components to earn full marks

5.3. **Deductions:**

5.3.1. Packaging errors:

(a) up to 10 marks for missing or incorrect Makefile

(b) up to 5 marks for missing or incorrect README

(c) up to 10 marks for failure to correctly separate code into header and source files

(d) up to 10 marks for missing documentation

5.3.2. Major design and programming errors:

(a) 50% of a marking component that uses global variables

(b) 50% of a marking component that consistently fails to use correct design principles

(c) 100% of a marking component that uses prohibited library classes or functions

(d) up to 100% of a marking component where Constraints listed are not followed

(e) up to 10 marks for bad style

(f) up to 10 marks for memory leaks

5.3.3. Execution errors:

(a) 100% of any marking component that cannot be tested because the code doesn't compile

(b) 50% of any marking component that cannot be tested because it doesn't execute in the course VM, or the feature is not used in the code, or data cannot be printed to the screen