Code in this exam was written on arch, I couldn’t get an debian 10 install up. Hopefully it works.

oppgave 1:

a:
C written by Dennis Ritchie, is most known for its use in Operating system and firmware applications, though it can be used for anything. It's general purpose and provies low level memory access. The lack of abstraction and garbage collectors makes the language fast. It’s mostly used for drivers and embedded systems.

b:
Linus Benedict Torvalds born 1969 in helsinki Finland. He is most known for creating The Linux Kernel and the git version control system. Linux is ran on most webservers in the world, git is the foundation of a lot of the online version control systems like gitBucket and Github. He’s also been an advocate for open source saying “for complex technical issues you really *need* open source simply because the problem space ends up being too complex to manage inside one single company. Even a big and competent tech company.”

(Linus Torvalds on Why Open Source Solves the Biggest Problems, December 21, 2021, The New Stack)

link: https://thenewstack.io/linus-torvalds-on-why-open-source-solves-the-biggest-problems/

c:

We have 4 steps when we compile a C program Preprocessing, Compiling, Assembling and Linking. When Preprocessing the compiler will remove Comments, it will turn macros into a assembly file for use, it will add the code from #include statements to the code. The Compiling step will turn C code into assembly code, it also checks the code for error or warnings. The Assembling step turns the assembly code into machine code (an Object file (.o)). The Linking will then include any library files. This acts like a dictonary for the computer to understand what function to call.

oppgave 2:

./oppgave2 filename.txt

Each file gotten from the task has had a .h file created for them and are built into a .o file.

You can see this In the make file. The program will read each line in the text file and remove formatting "/n/r" so we can compare them. The first line is what we'll compare to the rest and is malloc'd into char *first pointer for comparing later. The first word also cannot be compared to anything since it's the world we'll compare against. This means that .bAnagram and bDisjoint are both set to 0. When the first word is processed we'll compare call all function on all the words and write it to an sWord *list[100] Max words are 100 but this can be changed.

Each word get assigned an index based for many words are in the list. Since index should start at 1, we correct for this when adding the index by just +1. we malloc all structs to sizeof(sWord) + strlen(line) +1, to account for null termination. After that we strncpy the word onto the end of the struct this will malloc and free for us. When we have gone through each word and our list of sWord are done. We can start to write them to the .bin file (created if not there already). First we write the struct itself with sizeof the first element + iSize. We then write the word itself to the file. After the program is done we need to free the memory that we have malloced. This is done with the FreeStuctList() which loops through all elements of the list and frees it. After that we free char *first to close out the program.

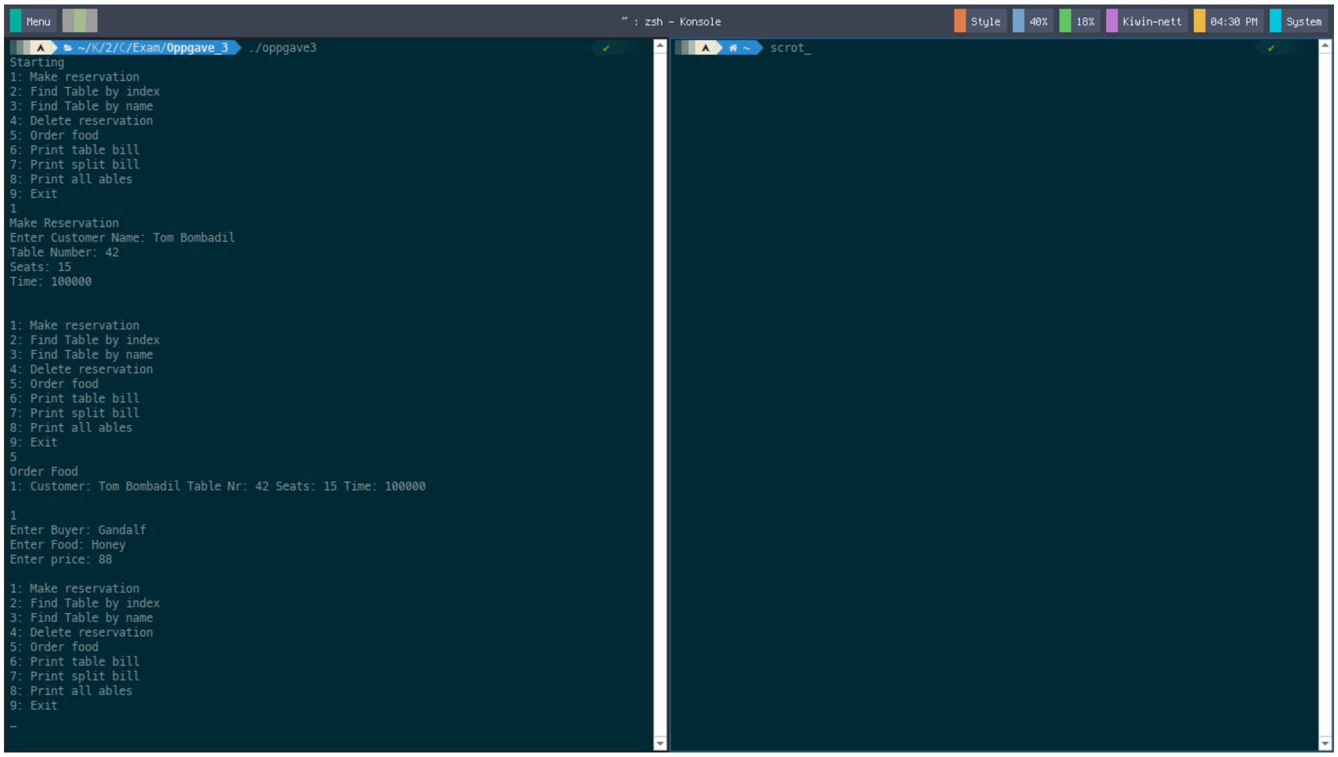
Output of "Hexdump -C Bin-word.bin" when using default word list
```
00000000  01 00 00 00 4a 55 4d 50  00 02 00 00 00 00 00 75  |....JUMP.......u|
00000010  6e 69 71 75 65 00 03 00  00 00 01 72 61 64 61 72  |nique......radar|
00000020  00 04 00 00 00 00 01 6c  69 73 74 65 6e 00 05 00  |.......listen...|
00000030  00 00 00 71 75 69 65 74  00 06 00 00 63 61 74 00  |...quiet....cat.|
00000040  07 00 00 00 00 01 53 49  4c 45 4e 54 00 08 00 00  |......SILENT....|
00000050  64 6f 67 00 09 00 00 00  01 00 00 64 65 69 66 69  |dog........deifi|
00000060  65 64 00 0a 00 00 45 4e  44 00                    |ed....END.|
0000006a
```

Oppgave 3

./oppgave3

table.h hold the defintions of the functions contained in table.c, the structs are also defined there and in the table.c file, I assume theres a way to only define them one place, but this works.

Main creates a Pointer to a table which is our head, we will pass this variable to all funcitons that require access to the list. Theres probably a better way of doing this, because I felt like I had to “drill” the variable through funtions to get it where I wanted. It goes from Main() → Menu() → MenuSwitchCase then its send to the functions that actually need the list. All functions that actually modify the list are in table.c file, there should be no changes from main, this should allow for easier refatoring later.


```
~/K/2/C/Exam/Oppgave_3  ./oppgave3
Starting
1: Make reservation
2: Find Table by index
3: Find Table by name
4: Delete reservation
5: Order food
6: Print table bill
7: Print split bill
8: Print all ables
9: Exit
1
Make Reservation
Enter Customer Name: Tom Bombadil
Table Number: 42
Seats: 15
Time: 100000


1: Make reservation
2: Find Table by index
3: Find Table by name
4: Delete reservation
5: Order food
6: Print table bill
7: Print split bill
8: Print all ables
9: Exit
5
Order Food
1: Customer: Tom Bombadil Table Nr: 42 Seats: 15 Time: 100000

1
Enter Buyer: Gandalf
Enter Food: Honey
Enter price: 88

1: Make reservation
2: Find Table by index
3: Find Table by name
4: Delete reservation
5: Order food
6: Print table bill
7: Print split bill
8: Print all ables
9: Exit
_
```
```
~  scrot_
```

When we exit we go though all tables and free all orders associated with each table before going to the next table.

Oppgave 4

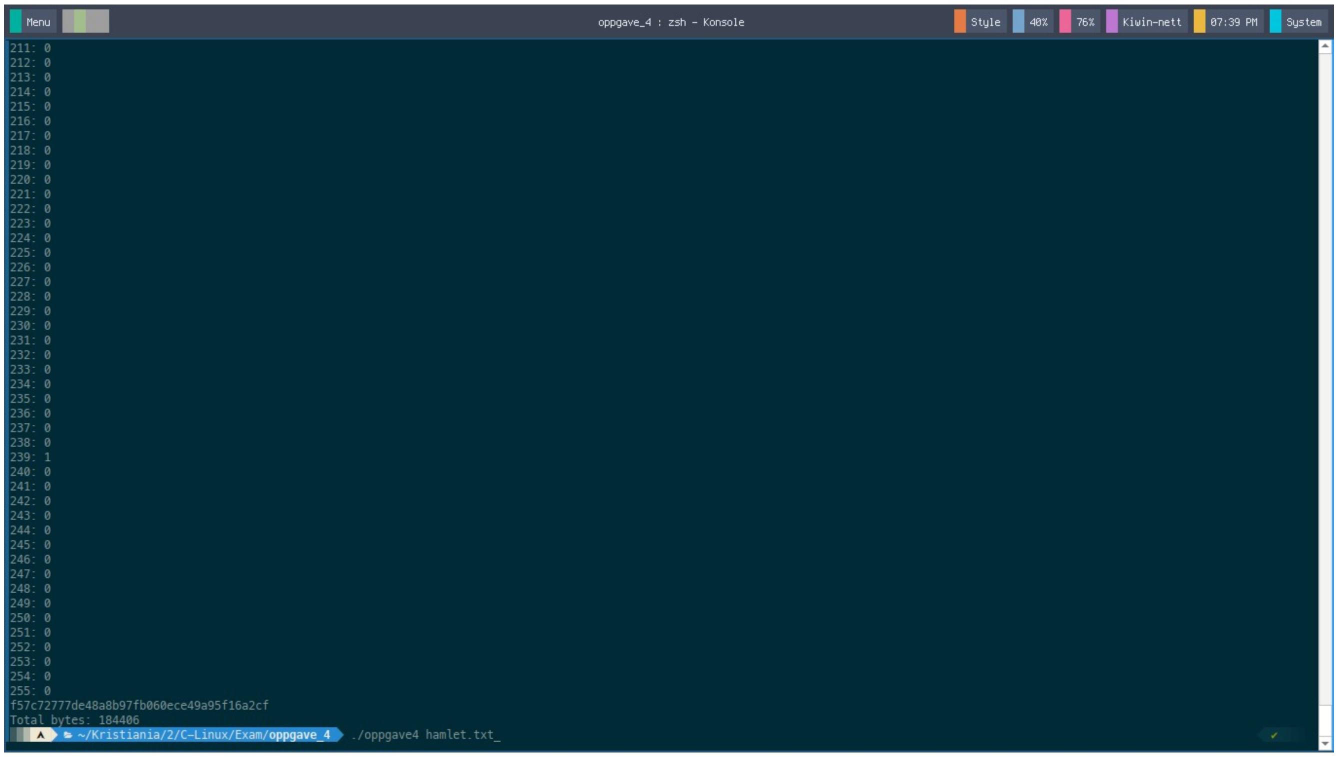
```
./oppgave4 hamlet.txt
```

Instead of passing in buffer directly, we just pass an struct Args as a void* and then cast it back to an Args struct in each thread. This will be our place for storing data both threads need.

we use 3 sem_t stores in the Args struct, mutex controls if the thread is allowed to use shared data. full signals that the buffer is full and can be counted by threadB. Empty signals that Thread A is allowed to read more from file. Full and Empty could just contain a number of how many bytes are in the buffer, but i thought it would be easier if they were binary.

When ThreadA reached EOF, it sets an EOF flag to 1 and breaks out of loop, this signals to ThreadB that, this is all thats left and breaks out of the loop.

When ThreadB starts it will call SHA1Init(&args→context) preparing for creating a sha1 hash. Each time buffer is full we call SHA1Update(&args→context, buffer, strlen(buffer)). When there is no more data to ready we call SHA1Final(&args→digest, &args→Context). We can then print the hash by printing each byte has hex.



```
211: 0
212: 0
213: 0
214: 0
215: 0
216: 0
217: 0
218: 0
219: 0
220: 0
221: 0
222: 0
223: 0
224: 0
225: 0
226: 0
227: 0
228: 0
229: 0
230: 0
231: 0
232: 0
233: 0
234: 0
235: 0
236: 0
237: 0
238: 0
239: 1
240: 0
241: 0
242: 0
243: 0
244: 0
245: 0
246: 0
247: 0
248: 0
249: 0
250: 0
251: 0
252: 0
253: 0
254: 0
255: 0
f57c72777de48a8b97fb060ece49a95f16a2cf
Total bytes: 184406
~/Kristiania/2/C-Linux/Exam/oppgave_4  ./oppgave4 hamlet.txt_
```

Oppgave 5:

This task is done as two folders, instead of one file. So we have client and server folder.
./server port1 port2
./server 4000 4343

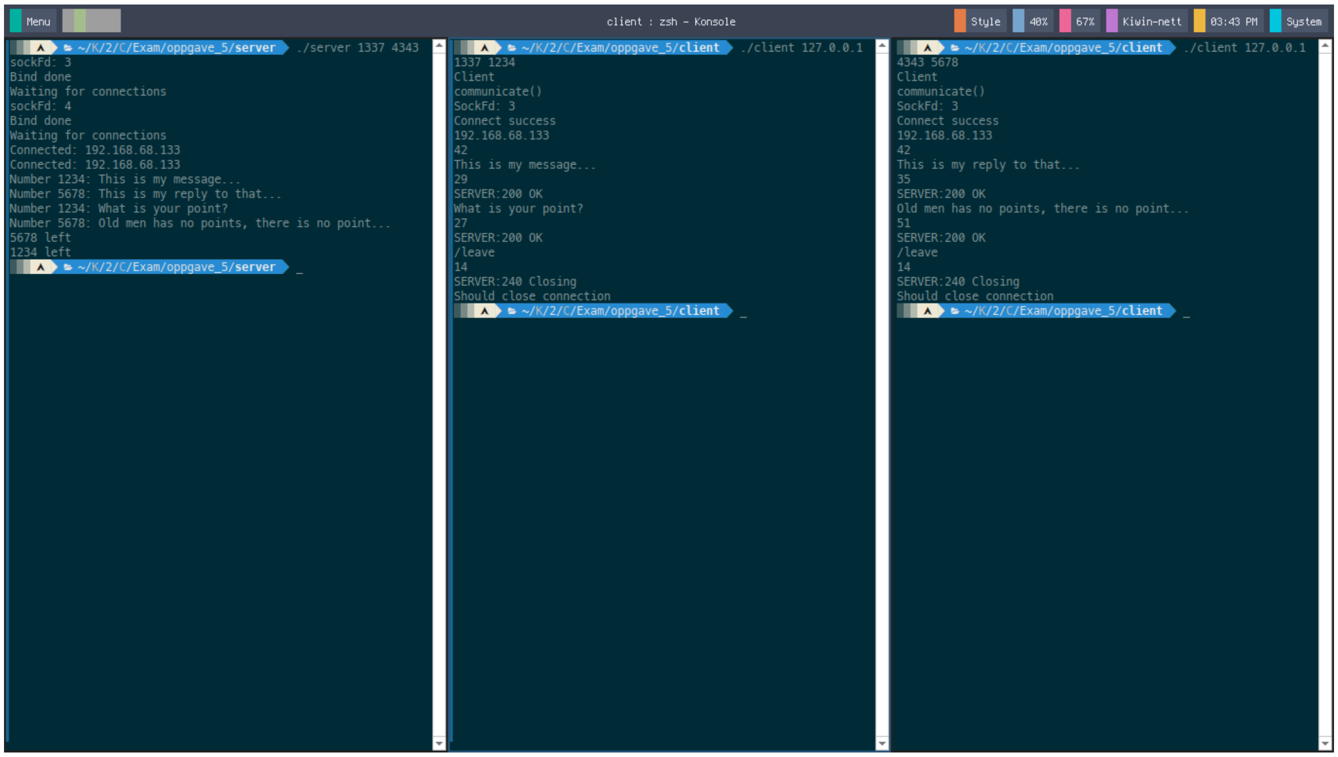
you can select whatever port you want

./client serverIp port phone
./client 127.0.0.1 4000 5678

client starts by sending a connection message containing Tlf, local ip, iMagicNumber and the size of the struct. Client connects with the server ip, port and a phone number. Server waits until the client has sendt this message, then checks that the client sendt the correct iMagicNumber, it then sets Targs→bConnected to 1, which starts the messaging loop. The loop is broken when the user types /leave. The server will only close once both clients has connected and have left.

From the SENDMESSAGE struct I removed the szMessage[] from the struct since I decided to send it right after in its own message. When the client sends a message, we send a struct containing the magic number and the size of struct + strlen(word). This allows us then on the server side to read the struct and store it. We can then do sizeof(struct) - iMessageSize to get the total amount of bytes we have to read to get the entire message from the client. I had some issues here with detecting the end of the string, but this was fixed by doing sanatizedMessage[bytes_read] = '\0'.

protocol: client connects and sends the connection struct, server response with the magicNumber. Then the client and the thread enters into a while loop allowing messeges to be sent back and forth. Client cannot send messages that don't have a length of 0, this was a hassle to fix on the server side, so I decided that the client couldn't send empty messages.



```
~/K/2/C/Exam/oppgave_5/server  ./server 1337 4343
sockFd: 3
Bind done
Waiting for connections
sockFd: 4
Bind done
Waiting for connections
Connected: 192.168.68.133
Connected: 192.168.68.133
Number 1234: This is my message...
Number 5678: This is my reply to that...
Number 1234: What is your point?
Number 5678: Old men has no points, there is no point...
5678 left
1234 left
~/K/2/C/Exam/oppgave_5/server  _
```

```
~/K/2/C/Exam/oppgave_5/client  ./client 127.0.0.1
1337 1234
Client
communicate()
SockFd: 3
Connect success
192.168.68.133
42
This is my message...
29
SERVER:200 OK
What is your point?
27
SERVER:200 OK
/leave
14
SERVER:240 Closing
Should close connection
~/K/2/C/Exam/oppgave_5/client  _
```

```
~/K/2/C/Exam/oppgave_5/client  ./client 127.0.0.1
4343 5678
Client
communicate()
SockFd: 3
Connect success
192.168.68.133
42
This is my reply to that...
35
SERVER:200 OK
Old men has no points, there is no point...
51
SERVER:200 OK
/leave
14
SERVER:240 Closing
Should close connection
~/K/2/C/Exam/oppgave_5/client  _
```