

2-week HOME EXAM

PG3401 C Programming

Permitted aids: All

Duration: 14 days

Grading scale/evaluation type: National grading scale A - F

Date: April 27th - May 11th, 2023

This is the exam text in English, see PG3401-Hjemmeeksamen-14dager-V23-NO.pdf for the Norwegian version of this text. All tasks in the exam set are the same in English and Norwegian versions, only language differs. You can hand in your exam in either English or Norwegian (Swedish and Danish is also accepted).

The exam text has 7 pages. There are a total of 6 tasks in the exam set.

There is a 2-week deadline for this home exam. The period may overlap with other exams you have, so it is important that you use the time efficiently and plan your time well, so that you do not sit right before the deadline and have to submit several exams at the same time. Please note that the exam **MUST** be submitted within the deadline set in Wiseflow, and the assignment can only be submitted via WISEFLOW. It will not be possible to submit the assignment after the deadline – this means that you should submit well in advance so that you can contact the exam office or support if you have technical problems.

It is emphasized that the student must answer the exam independently and individually, cooperation between students and plagiarism is not allowed. It's not allowed to present someone else's work as your own – this includes work done by artificial intelligence (such as text or code generation models). The exam must be solved on Linux.

Note that the tasks are made with increasing difficulty, and especially the last three tasks are more difficult than the first tasks. It is therefore encouraged to finish the first assignments (completely) so that the student does not spend all the time doing the last tasks first.

Format of submission

This is a practical programming exam (apart from task 1), so the focus should be on explaining how you have proceeded, justifying your choices and presenting any assumptions you have made in your solution.

If you are unable to solve a task, it is better if you hand in what you have done (even if it does not work), and explain how you have proceeded and what you did not achieve – than not answer the task at all. It is expected that everything works unless otherwise described in the PDF file, code that the student knows does not work

should also be commented as such in the code. If you know that the program is crashing, not compiling or not working as intended, it is important to explain this along with what steps you have taken to try to solve the problem.

The answer should be in 1 ZIP file, the name of the file should be PG3401_V23_[candidate number].zip. This file should have the following structure:
 task_2 \ makefile
 task_2 \ [...]
 [...]

In Wiseflow, you should upload a text answer with the name "PG3401_V23_[candidate number].pdf", and the ZIP file should be uploaded as an attachment to the text answer.

Be sure all files are included in the ZIP file. Each folder should have a makefile file, and no changes, third-party components or parameters should be required - the assessor will in shell on Debian Linux 10 go into the folder and type "make" and this should build the program with GCC.

The text answer should contain an answer to assignment 1 (write it to the point, does not need a long thesis). After the plain text answer, there should be 1 page of justification / documentation for each assignment, each of the documentation segments should be on a new page to make the text answer clear to the assessor. The answer must be in PDF format and have the correct file extension to be opened on both a Linux and a Windows machine (.pdf). Answers in other formats will not be read.

Task 1. Theory (5 %)

- a) Explain what the C programming language can be used for.
- b) Who is Dennis Ritchie and what is he known for in the field of Information Technology?
- c) Explain what the sudo command does in terminal on Linux, and what one typically uses this command for when using / managing Linux.

Task 2. File management and functions (15 %)

Download the following input-data file, it contains 10 integers (integer) where each number is on a line (there are line breaks between each number):

http://www.eastwill.no/pg3401/eksamen_v23_oppgave2.txt

Then download the following source files:

http://www.eastwill.no/pg3401/eksamen_v23_oppgave2_fib.c

http://www.eastwill.no/pg3401/eksamen_v23_oppgave2_prim.c
http://www.eastwill.no/pg3401/eksamen_v23_oppgave2_kvad.c
http://www.eastwill.no/pg3401/eksamen_v23_oppgave2_cube.c
http://www.eastwill.no/pg3401/eksamen_v23_oppgave2_perf.c
http://www.eastwill.no/pg3401/eksamen_v23_oppgave2_abun.c
http://www.eastwill.no/pg3401/eksamen_v23_oppgave2_def.c
http://www.eastwill.no/pg3401/eksamen_v23_oppgave2_odd.c

You should write a program that reads the numbers in the text file, for each of the numbers in the text file you should call the functions in the above source files to find out if the numbers are a) a number in the Fibonacci series, b) a prime number, c) a square number, d) a cubic number, e) a perfect number, f) a redundant number, g) a defective number, or h) an odd number.

The program should create an output file where each number is printed BINARY, along with metadata about the number according to the result from the functions listed above. Each number in the output file should be described with the following struct:

```
struct TASK2_NUMBER_METADATA {
    int iIndex; // The index of the number, first = 1
    int iNumber; // The number, as read from input file
    bool bIsFibonacci;
    bool bIsPrimeNumber;
    bool bIsSquareNumber;
    bool bIsCubeNumber;
    bool bIsPerfectNumber;
    bool bIsAbundantNumber;
    bool bIsDeficientNumber;
    bool bIsOddNumber;
}
```

You must create a makefile file (use makefile file taken from lecture 4, slide 65 – with the heading "Use this makefile") that links together the source files, and one or more header files for the program to compile correctly, make any changes required to the makefile and source files to get the program to build and run correctly.

Both the input file and the output file should be in the same directory as the program, and both files should be part of the submitted answer.

Task 3. List handling (20 %)

You will be creating a simple data structure to handle flight-departures. The "core" of the list should be a double-linked list of planes, each item (struct) in the list should contain pointers to both the previous item and the next item. The item should also contain a text string that is FLIGHTID (for example, BA-42), a text string describing DESTINATION, an integer for the number of SEATS, and an integer that should

serve as the TIME for departure. Each item should additionally contain a single-linked list of (reservations for) PASSENGERS.

Each item in the passenger list (reservations) should contain an integer containing the SEAT NUMBER, a text string with the NAME, and an integer containing the AGE of the traveler. The list should always be SORTED by seat number.

You will create functions that perform the following operations on the list:

- Add a flight to the list
- Add a passenger to a flight-departure (remember that the list should always be sorted by seat number)
- Retrieves a flight N from the list (counted from the start of the list, where the first item is item number 1) and prints on the screen all the data associated with the departure including the list of passengers
- Finds flight that matches departure TIME in the list, and returns what "item number" it has – re function above
- Deletes a flight (and all passenger reservations for the departure)
- Deleting a passenger's reservation (on a flight)
- Changing the seat of a passenger
- Search through the lists for a passenger's NAME (in all flights) and print on the screen all flights this passenger is associated with

You are going to create a main function that receives instructions from the user based on input from the keyboard, the main must be able to call all eight functions above (for example, in some form of menu) and ask the user for data necessary to call the mentioned functions. Main should clean up all data before it returns (an option in the menu must be to exit).

Task 4. Threads (15 %)

In practical programming, it is often efficient to lay out time-consuming operations in working threads, examples of which are file operations, network operations and communication with external devices. In this assignment, you will simulate such operations with a smaller data set – in order to save you time during the exam, a ready-made application has already been created for you to change.

Application consists of 3 threads, the main thread (running the main function) and 2 working threads. The main thread starts the working threads with mechanisms for

thread communication and synchronization (in this application, the main thread has no other function, and only starts the two threads that do the job themselves). The threads have a memory area with space of 4096 bytes for communication between the threads.

One working thread (thread A) should read a text file, work thread A should then send the file over to the other working thread (thread B) through multiple cycles using the memory space described above, signaling thread B that there is data available in the buffer. Thread B then count the number of instances of bytes with value 00, 01, 02, including; FF in the file it gets sent over. Thread A and thread B loop to process the file until it's finished. When the file is sent over in its entirety, work thread A will end. Worker thread B completes its count of bytes, and then prints to the terminal window the number of instances of each of the 256 possible byte values, before it also exits. The main thread waits for both threads to exit, cleans up correctly, and then closes application.

Download the following source file, this is a solution to the application as described:

http://www.eastwill.no/pg3401/eksamen_v23_oppgave4.c

Download a test file that can be used for this task (Hamlet by Shakespeare, taken from Project Gutenberg - <https://www.gutenberg.org/ebooks/2265>):

http://www.eastwill.no/pg3401/eksamen_v23_oppgave4_pg2265.txt

You will expand the code with the following changes:

- You must create a makefile file (use makefile file taken from lecture 4, slide 65 – with the heading "Use this makefile") to build the program, make any changes required to the makefile file and source file to make the program build and run correctly
- The program uses global variables, change this to that all variables are local variables in the main function and submitted as parameter to the two threads
- Thread A has hardcoded the name of the file to be read, change the program to take the file name as a parameter on the command line and passes this on to Thread A as a variable
- Instead of counting "byte values" (from 0 to 255), expand the functionality of counting letters and printable special characters (ascii code 32 to 126), you should also change the code that prints the number of instances on the screen to indicate which letter / character this is - for example, 'A' : ?
- In addition, count the occurrence of a couple of common short words; "and", "at", "it", "my", as well as the name "Hamlet", the number of instances should be printed on the screen after the letters of the previous point are printed
- Rewrite the code from using conditions to using semaphores
- Add code for explicit initialization of mutex and semaphores (with the *_init functions)
- Add comments to your code to document what the code does

Task 5. Network (25 %)

In this task, you will create a tool for setting up a reverse shell on a machine. A reverse shell is an application where a server sends commands to a client that the client executes "in terminal" as if a human had been sitting on the client and manually typing the same commands in the terminal window. The application should be able to be started both as a client and as a server, and you should create your own protocol for communication between client and server. When started as a server (for example, signaled by a parameter from terminal, which may be `-listen`) the application should open the specified port for LISTEN, for this task it is enough to bind to loopback on 127.0.0.1 (so as not to expose an open port remotely). It is recommended to use TCP.

The task should not be solved using Curl or other third-party libraries and should not be based on launching other applications in the operating system - only the use of Sockets as we have learned in lecture 10 on Network will give points on the task.

When started as a server, the application should BIND to a port the user selects, it is recommended for this task to listen only at address 127.0.0.1 so as not to expose the port outside your own machine. It is recommended to use TCP. Server application should be executed with port number as parameter from terminal, for example «task_5 -listen -port 42».

When started as client, the application should be executed with the server's IP address and port number from terminal, for example «task_5 -server 127.0.0.1 -port 42». When it starts, the application should CONNECT to the given port on the server process.

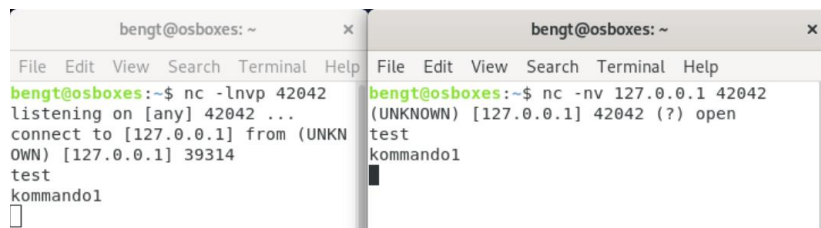
You will create a protocol for server-client traffic, at a minimum, it must include COMMAND as a text string sent to the client and RESULT as a text string sent back to the server. When a client is connected, the server application should accept input from the user in the form of text strings in the terminal, these should be Linux (terminal) commands to be sent to the client application in a format the student chooses. The client application should EXECUTE these commands and send the RESULT back to the server that prints this out on the screen. An example of code that does this (which you can use in the answer) can be found here:

http://www.eastwill.no/pg3401/eksamen_v23_oppgave5_exec.c

Both applications should be able to terminate with Ctrl-C, both applications should handle this (without crashing). When one application exits (either client or server), the other application should also exit.

A tip for testing server/client applications is to open two terminal windows in Linux and launch one instance "task_5 -listen -port 42" in one window and the other instance "task_5 -server 127.0.0.1 -port 42" in the other window. Example (from the tool netcat which is an existing tool that can do the same thing that their task should

do):



The image shows two terminal windows side-by-side. The left window is titled 'bengt@osboxes: ~' and shows a Netcat listener on port 42042. It receives a connection from 127.0.0.1 and the user 'kommando1' sends the text 'test'. The right window is also titled 'bengt@osboxes: ~' and shows a Netcat client on 127.0.0.1 port 42042. It connects to the listener and sends the text 'test'.

```
bengt@osboxes: ~  
File Edit View Search Terminal Help  
bengt@osboxes:~$ nc -lnvp 42042  
listening on [any] 42042 ...  
connect to [127.0.0.1] from (UNKN  
OWN) [127.0.0.1] 39314  
test  
kommando1  
█
```

```
bengt@osboxes:~$ nc -nv 127.0.0.1 42042  
(UNKNOWN) [127.0.0.1] 42042 (?) open  
test  
kommando1  
█
```

Oppgift 6. File management and text parsing (20 %)

You're creating an application that acts as a "code beautifier," an application that changes source code to something that fits the author's coding style (making the code "prettier").

The application should take a filename to a C-source file as a parameter when started from terminal. The application should read this file and make 3 changes to the file then save it with the same name but added `_beautified` before `.c` in the filename.

A) First, the application should take all instances of while-loops and replace these with for-loops, that means code like this:

```
a = 0;  
while (a < b) {... a++; }
```

should be replaced with one for loop and will look something like this:

```
for (a = 0; a < b; a++) {...}
```

Where... indicates the rest of the code in the loop, and naturally must be kept as is. It can be assumed that only "simple" loops (which should have been a for-loop) with a variable set to a value just before the loop and the loop has a simple test with this value (as in the example above) are replaced - and all other uses of while are kept as is.

B) The application will then force the use of Hungarian notation in the code, this will be solved by finding all variables of the type "unsigned int" and renaming all variables of that type to be "uiAbc" where Abc is the variable's original name, for example, "unsigned int counter" will result in all instances of the variable being changed to "uiCounter". (You shouldn't do this for other types, as it'll be too much work.)

C) In addition, all instances of 3 spaces should be replaced with a tab (ASCII code 0x09).

+

End of task set