Технически университет – София

Факултет Компютърни системи и технологии

|  |  |
| --- | --- |
| Изготвил: | Възложил: |
| Иван Ивайлов Милев | Д-р инж. Десислав Андреев |
| Дата на защита:  25.05.2021 |  |

Курсова работа

Тема №29

# Анализ на изготвеното приложение

Условие №29:

Програмите за бенчмаркинг са изключително необходими при оценката на дадена

функционалност. За тази цел ни трябва приложение, което да реализира функцията

sort(<params>), но зад нея стоят различни реализации. Намерете и използвайте поне

четири готови алгоритъма за сортиране (напр: бърза сортировка, сортировка на

мехурчето, сортировка на Шел, сортировка чрез сливане). Пазете резултатите на

алгоритмите в контейнер и при извикването на sort(<params>) го обновявайте.

Подгответе файлове с данни, с които да тествате всеки алгоритъм (малко/много на

брой сортирани/несортирани числа). Записвайте резултатите (структура по ваш избор,

съдържаща основните характеристики на измерването) на всеки тест в друг файл (със

структурата tm от time.h засичате колко време е отнело). Този алгоритъм, който се е

справил най-добре на най-много тестовете бива отпечатан. Този, който се справи най-

зле - бива изтрит от контейнера.

Изготвената програма представлява конзолно приложение състоящо се от два основни компонента – сървър и клиент. Потребителят трябва да може да избира как да го стартира като сървър или съответно като клиент. В клиентската си част приложението ще дава възможност на потребителя да въведе командата sort <params> като параметрите ще са първо изредени със спейс имена на файлове с тестови данни след това последвани от флага –а, който ще обозначава началото на изреждането на алгоритми, с които да бъдат сортирани тестовите данни. След като въведе тази команда потребителя и натисне ентър всички нужни данни заедно със самата команда ще бъдат изпратени на сървъра, където ще се изпълни бенчмарка и след завършването ще се върне резултат на клиента.

Резултата ще представлява представянето на различните алгоритми във всяка итерация на тестване(итерациите на тестване са различните тестови данни, които потребителя е предоставил). Като след всяка итерация най-бавно справилия се алгоритъм ще отпада от контейнера(от алгоритмите, с които се тества). В края на потребителя ще бъдат принтирани всички итерации с всички алгоритми за съответната итерация.

# Функционално описание на приложението

Приложението е разделено на две основни части – Сървър и клиент. Връзката между тях се осъществява посредством сокети.

## Сървър

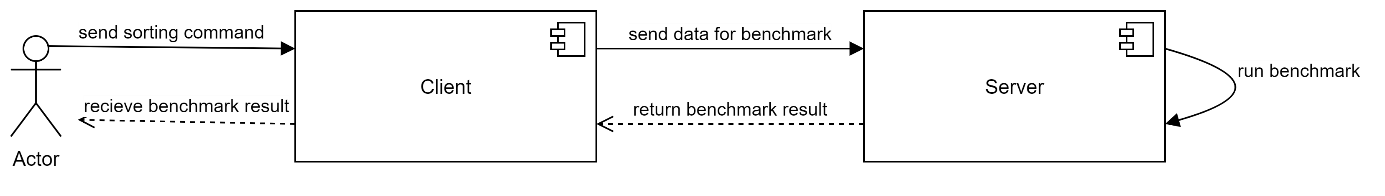
Сървъра има три основни функционалности-

1. Връзка с клиента така че да получи нужните му данни, за да започне бенчмарка
2. Изпълнение на бенчмарка (изпълняване на различните алгоритми за сортиране и запис на представянето)
3. Изпращане обратно на клиента резултата от изпълнения бенчмарк.

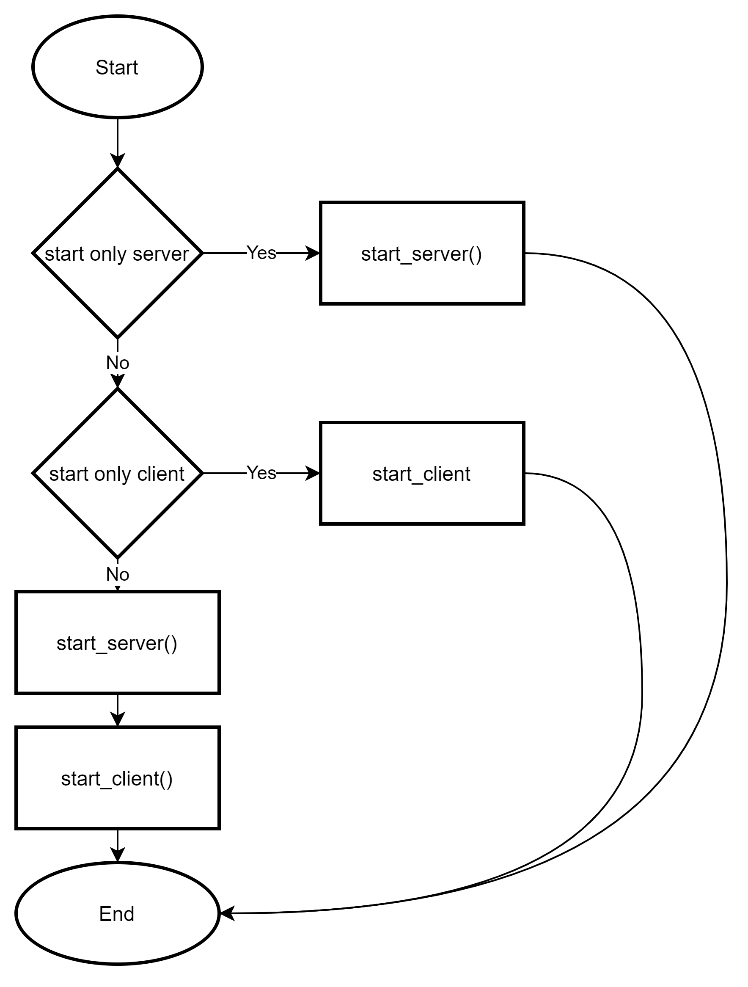
## Клиент

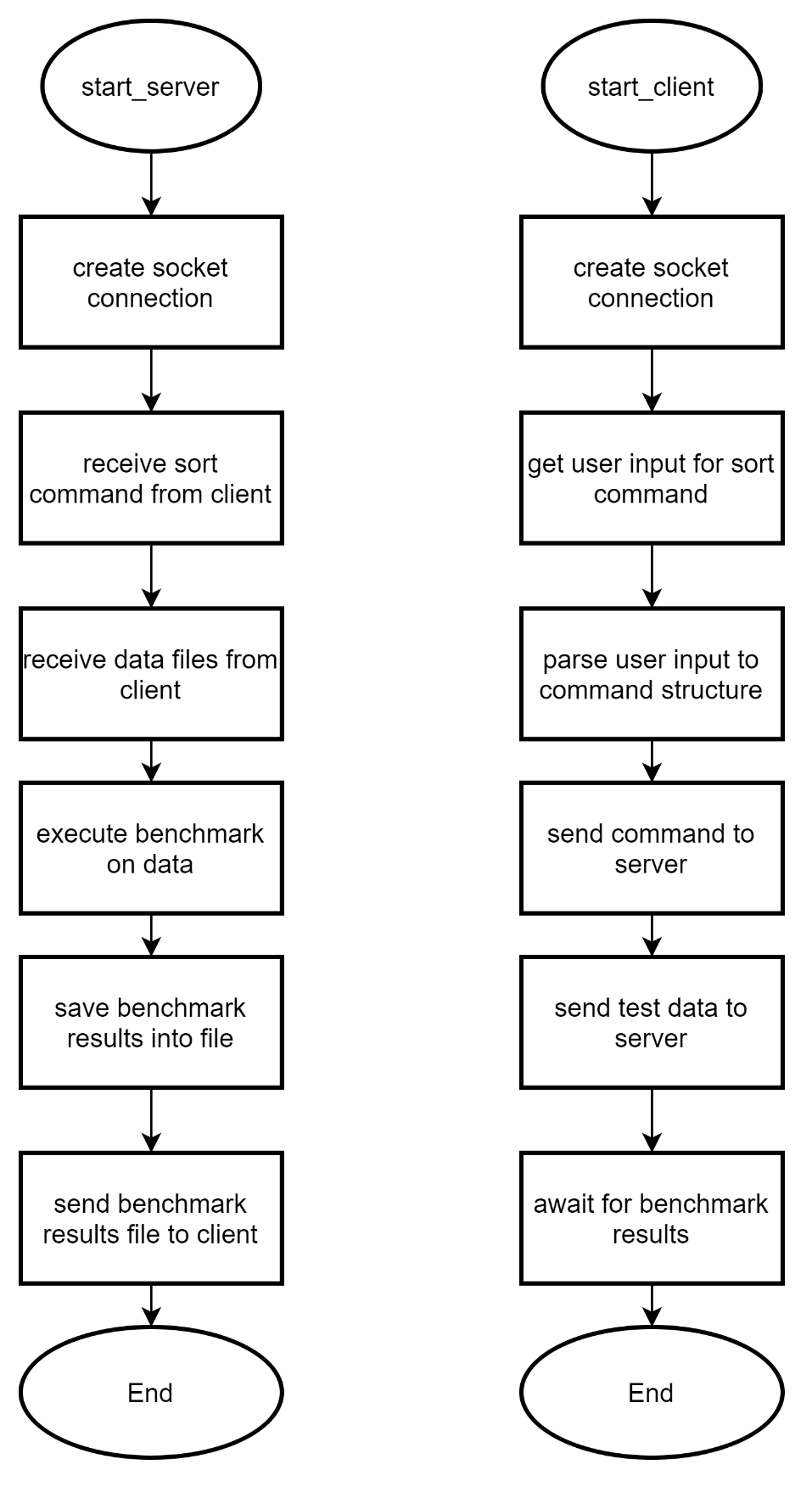
1. Прочитане на командата от потребителя и привеждането и във вид на структура.
2. Изпращане на всички данни към сървъра (имена на алгоритми и файлове с тестови данни) нужни за бенчмарк.
3. Получаване на файла с резултат от бенчмарка и принтирането му на стандартния изход.

## Диаграма на високо ниво:

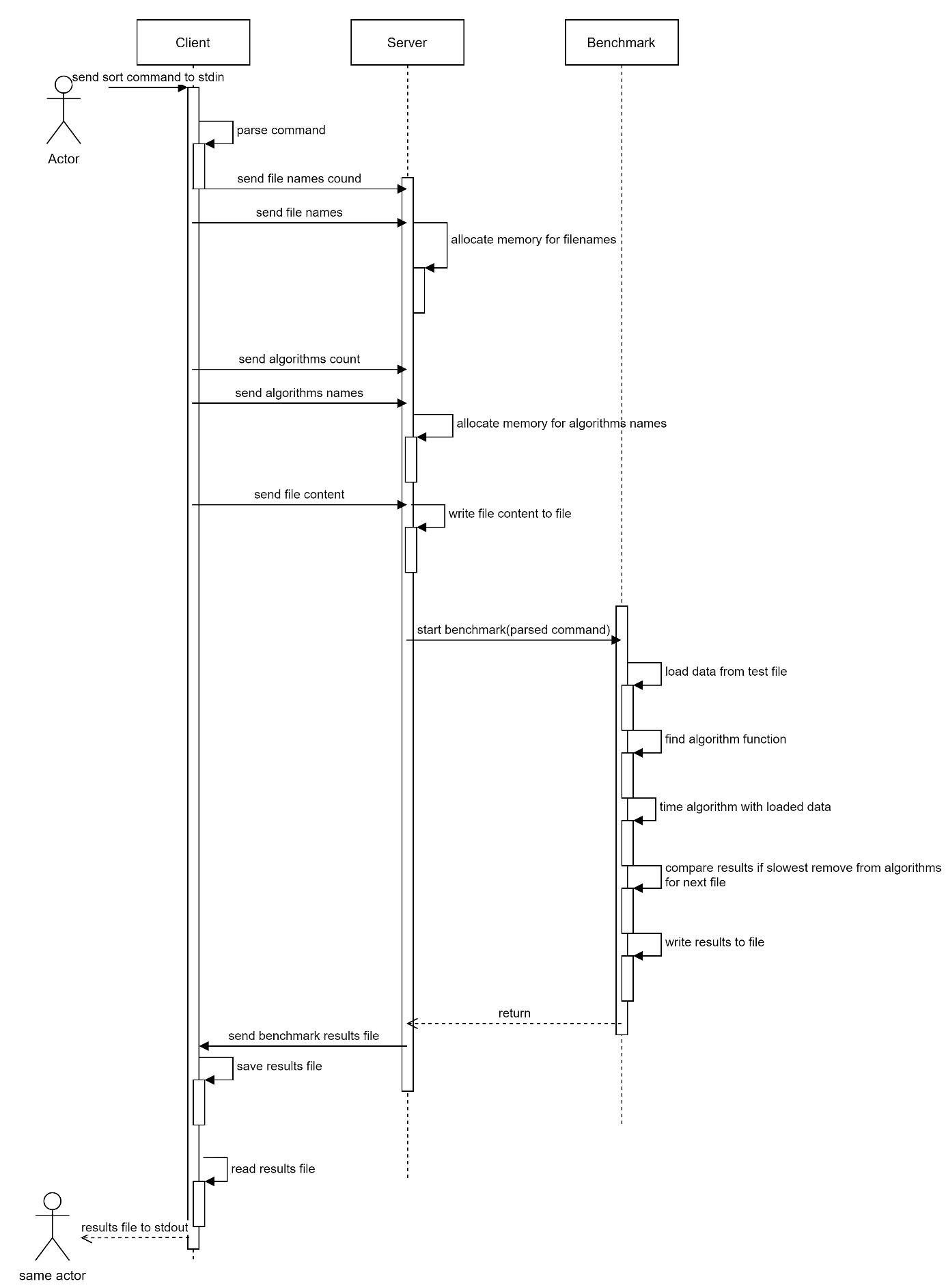


## Диаграма на последователността:





## Диаграма на последователностите:



# Изпълнение на функционалностите

## Сървър(coursework.c):

### void start\_server()

Стартира сървъра, като създава сокет и започва да чака да получава данни.  
Не приема параметри и не връща стойност

### int load\_array(char \* filename, int \*\* array)

Зарежда съдържанието на файл с име filename в масив с указател към него int \*\* array

char \* filename – името на файлът, от който ще се зареждат данните

int \*\* array – указател към масивът, в който данните ще се запишат

Връща броя числа, които са записани в array(размера на array)

### double benchmark\_sort(void(\*f)(int \*, int), int \* array, int array\_size)

Изпълнява бенчмарк за подадения му като параметър алгоритъм за сортириране.

void(\*f)(int \*, int) – функцията за сортиране, за която ще бъде засечено време

int \* array – масивът, който ще бъде сортиран от функцията за сортиране

int array\_size - размерът на масива, койтп ще бъде сортиран

Връща времето, за което масивът е бил сортиран в секунди

### void \* get\_algorithm(char \* algorithm\_name)

Дава указател към функция за сортиране отговаряще на името на сортировката.

char \* algorithm – името на алгоритъма, за който се търси фунцкия

Връща указател към функцията за сортиране, която отговаря на даденото име. Ако не е намерена такава връща void.

### void execute\_benchmark(struct parsed\_command \* cmnd)

Изпълнява цялостния бенчмарк за дадена команда.

struct parsed\_command \* cmnd – указател към обработената команда

### void listen\_for\_client(int sock\_fd, struct sockaddr\_in addr\_con, int addrlen)

Чака данни от клиента – да започне да му изпраща обработена потребителска команда.  
int sock\_fd – дескриптор на сокета

struct scokadd\_in addr\_con – адрес на дестинацията

int addrlen – размерът на адрес на дестинацията

### void receive\_files(char \*\* filenames, int size, int sock\_fd, struct sockaddr\_in addr\_con, int addrlen)

Функцията очаква да прочете съдържанието на size на брой файлове, като всичките ще бъдат записани в съответен файл с име от filenames.

char \*\* filenames – масив от имената на файловете

int size – брой имена на файлове

int sock\_fd – дескриптор на сокета

struct scokadd\_in addr\_con – адрес на дестинацията

int addrlen – размерът на адрес на дестинацията

### void send\_benchmark\_data(int sock\_fd, struct sockaddr\_in addr\_con, int addrlen)

Изпраща на клиента файлът с резултата от бенчмарка.

int sock\_fd – дескриптор на сокета

struct scokadd\_in addr\_con – адрес на дестинацията

int addrlen – размерът на адрес на дестинацията

### void write\_sort\_result(int \* fd, struct sort\_res result)

Записва в двоичен файл с дескриптор fd структурата result

int \* fd – файлов дескриптор на файла, в който се пише

struct sort\_res result – структура, която да бъде записана

### int in\_slowest(char \* algoname, char \*\* algos, int size)

Проверява дали algoname се съдържа в algos. Проверката става по адрес, а не по стойност.  
char \* algoname – името на алгоритъма, който търсим

char \*\* algonames – масив от имена на алгоритми, в който търсим

int size – размера на algonames

Връща 1 ако е намерен и 0 ако не е

## Клиент(client.h):

### void start\_client()

Стартира клиента – изписва информативни съобщения, отваря връзката със сокета и чака потребителя да въвжеда команди.

### void send\_data(struct parsed\_command cmnd, int sock\_fd, struct sockadd\_in addr\_con, int addrlen);

Изпраща командата както и файловете за тестване по сокета.

struct parsed\_command cmnd – командата, която ще бъде изпратена

int sock\_fd – дескриптор на сокета

struct scokadd\_in addr\_con – адрес на дестинацията

int addrlen – размерът на адрес на дестинацията

### void receive\_results (int sock\_fd, struct sockadd\_in addr\_con, int addrlen);

Очаква да получи от сокета данните на файла с резултата от бенчмарка, те ще бъдата записани във файл с име benchmark\_results.

int sock\_fd – дескриптор на сокета

struct scokadd\_in addr\_con – адрес на дестинацията

int addrlen – размерът на адрес на дестинацията

### struct parsed\_command \* parse\_input(char \* command);

Ще обработи въведения от потребителя низ(command) и ще върне обработения резултат, като потребителя на функцията трябва да се погрижи да освободи паметта заделена за обработената команда.

char \* command – низ съдържащ командата въведена от потребителя.

Връща обработената команда

### void show\_results(char \* filename);

Принтира на стандартния изход резултатите от бенчмарка, който е съхранен във файл с име filename.

char \* filename – името на файла с бенчмарка

### void free\_command\_memory(struct parsed\_command \* cmdn);

Освобождава паметта заделена за струтурата за обработена команда

struct parsed\_command \* cmnd – командата, която ще бъде освободена от паметта.

## Допълнителни методи около сокети (socket\_utils.h)

### void send\_string(char \* string\_to\_send, int sock\_fd, struct sockaddr\_in addr\_con, int addrlen);

Изпраща дадения низ чрез сокета.

char \* string\_to\_send – низът, който ще бъде изпратен

int sock\_fd – дескриптор на сокета

struct scokadd\_in addr\_con – адрес на дестинацията

int addrlen – размерът на адрес на дестинацията

### void send\_file(char \* filename, int sock\_fd, struct sockaddr\_in addr\_con, int addrlen);

Изпращане съдържанието на файл с име filename чрез сокета.

char \* filename – името на файлът, чието съдържание ще бъде изпратено.

int sock\_fd – дескриптор на сокета

struct scokadd\_in addr\_con – адрес на дестинацията

int addrlen – размерът на адрес на дестинацията

### void receive\_file(char \* filename, int sock\_fd, struct sockaddr\_in addr\_con, int addrlen);

Очаква изпращането на данни съдържани във файл, тези данни ще бъдат записани във файл с името filename. Данни се записват във файла до достигане на EOF или ‘\0’

char \* filename - името на файла, в който ще се запишат получените данни

int sock\_fd – дескриптор на сокета

struct scokadd\_in addr\_con – адрес на дестинацията

int addrlen – размерът на адрес на дестинацията

### void receive\_string(char \*\* array\_to\_hold, int sock\_fd, struct sockaddr\_in addr\_con, int addrlen);

Очаква да получи низ от сокета, който ще бъде записан в array\_to\_hold.

char \*\* array\_to\_hold – указател към низът, в който ще бъде записан резултата

int sock\_fd – дескриптор на сокета

struct scokadd\_in addr\_con – адрес на дестинацията

int addrlen – размерът на адрес на дестинацията

## Допълнителни методи за работа с масиви и файлове(utils.h)

### int write\_to\_file(int \* fd, char \* buffer)

Записва символите от buffer във файла с дескриптор fd докато не стигне до EOF или ‘\0’.  
int \* fd - файлов дескриптор

char \* buffer – низът, в който се записва

Връща 0 ако записаният резултат е по-малък от BUFFER\_SIZE, ако ли не 1

### int read\_file\_buf(int \* fd, char \* buffer)

Записва символите от файла с дескриптор fd в buffer докато не стигне до EOF или ‘\0’.  
int \* fd - файлов дескриптор

char \* buffer – низът, в който се записва

Връща 0 ако EOF или ‚\0‘ е прочетено , ако ли не 1

### int write\_to\_buffer(int indx, char \* read\_from, char \* buffer)

Записва символите от read\_from низа в buffer низа докато не стигне до EOF или ‘\0’.

int indx - индекс от който да започне нда чете read\_from

char \* read\_from – низът, от който четем

char \* buffer – низът, в който се записва

Връща нова стойност на indx, такава че да може да се продължи четенето в бъдеще от докъдето е стигнато, ако е стигнато до EOF или ‚\0‘ връща -1

### int read\_from\_buffer(int indx, char \* buffer, char \*array)

Записва символите от read\_from низа в buffer низа докато не стигне до EOF или ‘\0’.

int indx - индекс от който да се записва в array

char \* buffer – низът, от който четем

char \* array – низът, в който се записва

Връща нова стойност на indx, такава че да може да се продължи записа в бъдеще от докъдето е стигнато, ако е стигнато до EOF или ‚\0‘ връща -1

## Сортировки (sorting.h)

### void swap(int \* a, int \* b)

Разменя двете променливи

int \* a – указател към първата променлива, която трябва да се размени

int \* b – указател към втората променлива, която трябва да се размени

### void bubble\_sort(int \* array, int size)

Сортира по метода на мехурчето масива array

int \* array – масива, който ще бъде сортиран

int size –размера на масива, който ще бъде сортиран

### void quick\_sort\_helper(int \* array, int start, int end)

Сортира по метода на бързата сортировка масива array(рекурсивно)

int \* array – масива, който ще бъде сортиран

int start – начален индекс за бързата сортировка

int end – краен индекс за бързата сортировка

### void quick\_sort (int \* array, int size)

Сортира по метода на бързата сортировка – извиква quick\_sort\_helper. Изпозлзва се, за да се запазят параметрите „интерфейса“ както и на всички останали.

int \* array – масива, който ще бъде сортиран

int size –размера на масива, който ще бъде сортиран

### void selection\_sort(int \* array, int size)

Сортира по метода на пряката селекция на масива array

int \* array – масива, който ще бъде сортиран

int size –размера на масива, който ще бъде сортиран

### void heap\_sort (int \* array, int size)

Сортира по метода на пирамидалното соритране на масива array

int \* array – масива, който ще бъде сортиран

int size –размера на масива, който ще бъде сортиран

### void stable\_selection\_sort (int \* array, int size)

Сортира по метода на пряката селекция(устойчив вариант) на масива array

int \* array – масива, който ще бъде сортиран

int size –размера на масива, който ще бъде сортиран

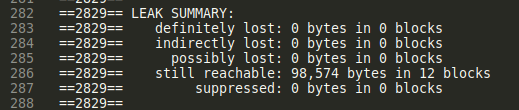
# Експериментални данни

Всички експериментални данни са генерирани с generateTestFiles.py - <https://github.com/ivanmilevtues/SystemProgramming/blob/main/generateTestFiles.py>

И е тествано с файлове от 0 до 5 което значи с масиви от данни съответно с 0, 10, 100, 1000, 10000, 100000 елемента. В последният генериран файл има 1000000 записа.

Анализ на heap с Valgrind и top:

Голяма част от паметта се заделя динамично в приложението особено в сървърната част на приложението. След изпълнението на програмата се вижда, че по-голямата част от паметта се освобождава, реално има данни, които не се освобождават, но не е загубена референция към тях. Което означава, че при правилен изход от програмата и добро разбиране от програмиста трябва да бъдат отстранени.



Анализ на Compile-time data:



text – размера на кодовия сегмент – това което попада във Flash паметта.

data – размера на данните които са инициализирани в кода например filename[128] = “test” и различните int x = 0

bss – размер на блока стартиран от символен сегмент, което попада в RAM паметта.

dec = text + data + bss = 15895 = 3e17(hex)

# Приложение – изходен код на приложението

<https://github.com/ivanmilevtues/SystemProgramming>