

# Käyttöjärjestelmä ja systemiohjelmointi

## Project documentation.

Written by: Otto Oikarinen ([otto.oikarinen@student.lut.fi](mailto:otto.oikarinen@student.lut.fi))

Last edited: 13.5.2024

Source code available:

<https://github.com/OttoOikarinen/OperatingSystemsCourseProject/>

Code has been written by me. ChatGPT has been used for explaining different functions and helping to debug the code.

## Project 1: reverse.c

This is the Project 1: Warmup to C and Unix programming.

It consist of 1 file, reverse.c which includes the whole program.

The goal of the program is to reverse lines in a given input source and print them to given output. Inputs can be taken from a file or from terminal. Output can be another file or the terminal.

The file includes main-function and several other functions, which together handle the program. Main function is mainly concerned with ensuring the program has been called properly (only maximum two parameters after the name of the program) and deciding correct inputs and outputs. Then it calls reverseFile-function which takes in the input and output.

reverseFile-function is the function which is responsible with reading the inputs, taking the lines and storing them to a linked list. This is done with the help of

getline, malloc and other basic commands. There is a specific struct called LIST, which stores pointers to previous and next nodes as well as pointer to the line.

After storing the lines to a list, printList-function is called. It travels to the end of the list and then back to the beginning while printing all lines to the chosen output.

After that memory is freed in freeList-function and main function returns 0.

## **Project 2:**

Project 2 includes programs my-cat, my-grep, my-zip and my-unzip. This is documentation for them.

### ***my-cat.c***

my-cat.c is a simple application which purpose is to print the files which are given in the parameters. For example calling:

```
./my-cat file1 file2 file3
```

would print the insides of file1, file2 and file3. If no parameters are given, my-cat will just exit without error.

The program consists of two functions:

main-function is responsible for checking, if there are any parameters. If parameters are found, it will open the file and call printFile-function. After that, it will close the file and open another one, if there are more parameters left. If not, it will stop the program.

printFile-function is actually responsible for the printing part. It reads a line with getline() and then prints it immediately after that. It stops when all the files have been read.

```
otto@rivendell:~/Documents/Yliopisto/Käyttöjärjestelmä/CourseProject$ ./my-cat test.txt
iiiiww 882828288888 akskjkjkjjjjkkklslsksj
searchterm
yes
yes
searchterm
joooooooooo
kyllä toimii vain hyvin
f
otto@rivendell:~/Documents/Yliopisto/Käyttöjärjestelmä/CourseProject$
```

## ***my-grep.c***

my-grep is used to find a searchterm from the input given to it. The inputs can be files or it can be the standard input. The program is called for example:

```
./my-grep searchterm [file ...]
```

The searchterm is mandatory, without it the main-function will print advice to using the program and exit with error. The file-part is voluntary and there can be as many files as the user wishes.

The program consists of two functions:

main-function is responsible for checking the parameters. If there are none, it will give advice and exit with error. If there is only one parameter (so a searchterm), it will call the grepFile-function and give it stdin as input and the only parameter as a cSearchTerm. If there are more parameters, the function will try to open them and pass them as inputs to the grepFile-function one by one.

The grepFile-function is responsible for searching for the term. If the input is stdin, it prints information to the user, that they can stop the program by writing 'quit'. It will read a line from the input using getline(). Then it will search the line for the searchterm using strstr(). If a match is found, it will be printed. Function returns when no more lines can be found or user writes 'quit'.

```
otto@rivendell:~/Documents/Yliopisto/Käyttöjärjestelmä/CourseProject$ ./my-grep searchterm test.txt
searchterm
searchterm
otto@rivendell:~/Documents/Yliopisto/Käyttöjärjestelmä/CourseProject$
```

## **my-zip.c**

my-zip is used to compress one or more files. It gets the files as parameters when running the program. The program is called like:

```
./my-zip file1 [file2 ...]
```

One file is mandatory, but users can use as many files as they wish.

The program consists of two functions:

main-function is responsible for checking the parameters. If no files are specified when running the program, it will print advice about the usage of the program and then exit with error. If there are one or more files, the function will try to open them and then pass them as an input to the zipFile-function. After that, the function will close the file and open another one, if there are parameters left.

The zipFile-function is responsible for compressing the file. If the file is empty, it returns nothing. Otherwise it will read characters from the file. If there are multiple same characters back to back, it will count the characters. Then when there are no more same characters, it will print the count and the character.

```
otto@rivendell:~/Documents/Yliopisto/Käyttöjärjestelmä/CourseProject$ ./my-zip test.txt
iw 8282828 akskjkjkjklslksj
searchterm
yes
yes
searchterm
o
kylöö toimi vain hyvin
f
otto@rivendell:~/Documents/Yliopisto/Käyttöjärjestelmä/CourseProject$
```

## **my-unzip.c**

my-unzip is used to decompress the files compressed with my-zip. It gets files as parameters when starting the program. The program is called like:

```
./my-unzip file1 [file2 ...]
```

One file is mandatory, but users can use as many files as they wish.

The program consists of two functions:

main-function is responsible for checking the parameters. If no files are specified when running the program, it will print advice about the usage of the program and then exit with error. If there are one or more files, the function will try to open them and then pass them as an input to the unzipFile-function. After that, the function will close the file and open another one, if there are parameters left.

The unzipFile-function is responsible for decompressing the data. It will first read a number and then a character. Then it will print as many characters as the number tells is. This continues as long as there are new characters.

```
otto@rivendell:~/Documents/Yliopisto/Käyttöjärjestelmä/CourseProject$ ./my-zip test.txt > encoded.z
otto@rivendell:~/Documents/Yliopisto/Käyttöjärjestelmä/CourseProject$ ./my-unzip encoded.z
iiiiww 882828288888 akskjkjkjjjjkkklslkskj
searchterm
yes
yes
searchterm
jooooooooo
kyllä toimii vain hyvin
f
otto@rivendell:~/Documents/Yliopisto/Käyttöjärjestelmä/CourseProject$
```

## Project 4: Kernel Hacking

To add a system call for getreadcount() to this xv6-kernel, I made the following changes to the code:

In syscall.c line 106:

```
104  extern int sys_write(void);
105  extern int sys_uptime(void);
106  extern int sys_getreadcount(void);
107
108
109  static int (*syscalls[])(void) = {
110      [SYS_read] = sys_read,
```

and line 131:

```
129  [SYS_mkdir] = sys_mkdir,
130  [SYS_close] = sys_close,
131  [SYS_getreadcount] = sys_getreadcount,
132
133  };
134
135  void
```

In syscall.h line 23:

```
20  #define SYS_link 19
21  #define SYS_mkdir 20
22  #define SYS_close 21
23  #define SYS_getreadcount 22 // For assignment
24
```

In sysproc.c line 94-98:

```
94 // This whole function was added for assingment
95 // It returns the readcount.
96 int sys_getreadcount(void) {
97     return myproc()->readcount;
98 }
99
```

In proc.h line 52:

```
49 struct file *ofile[NOFILE]; // Open files
50 struct inode *cwd;           // Current directory
51 char name[16];               // Process name (debugging)
52 int readcount;               // For assignment task.
53 };
```

In usys.S line 32:

```
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(getreadcount)
33
```

In user.h line 26:

```
23 char *sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int getreadcount(void);
27
28
```

I then added a new file getreadcount.c:

It is the function that the user is able to call to check what the read count is at the moment. It is accessible on the command line using 'getreadcount'.

It has the following code:

```
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 int main(int argc, char *argv[]) {
6
7     printf(1, "Read count at the moment is: %d\n", getreadcount);
8     exit();
9 }
```

## Running xv6 kernel:

You can run the xv6 kernel by downloading the source code from my CourseProject repository. Then you can run the kernel by using terminal and command: *'sudo make qemu'*.

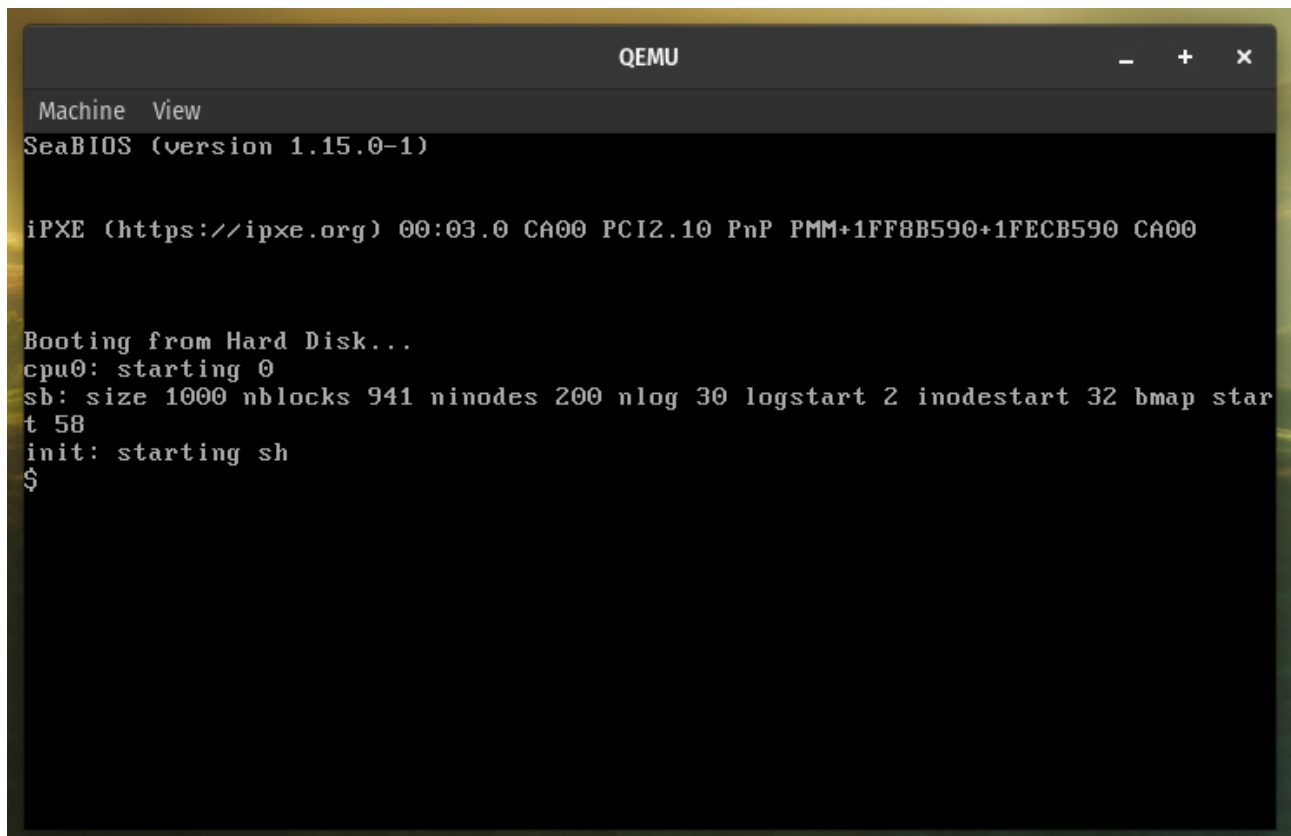
This requires you have qemu installed.

Photos of the process:

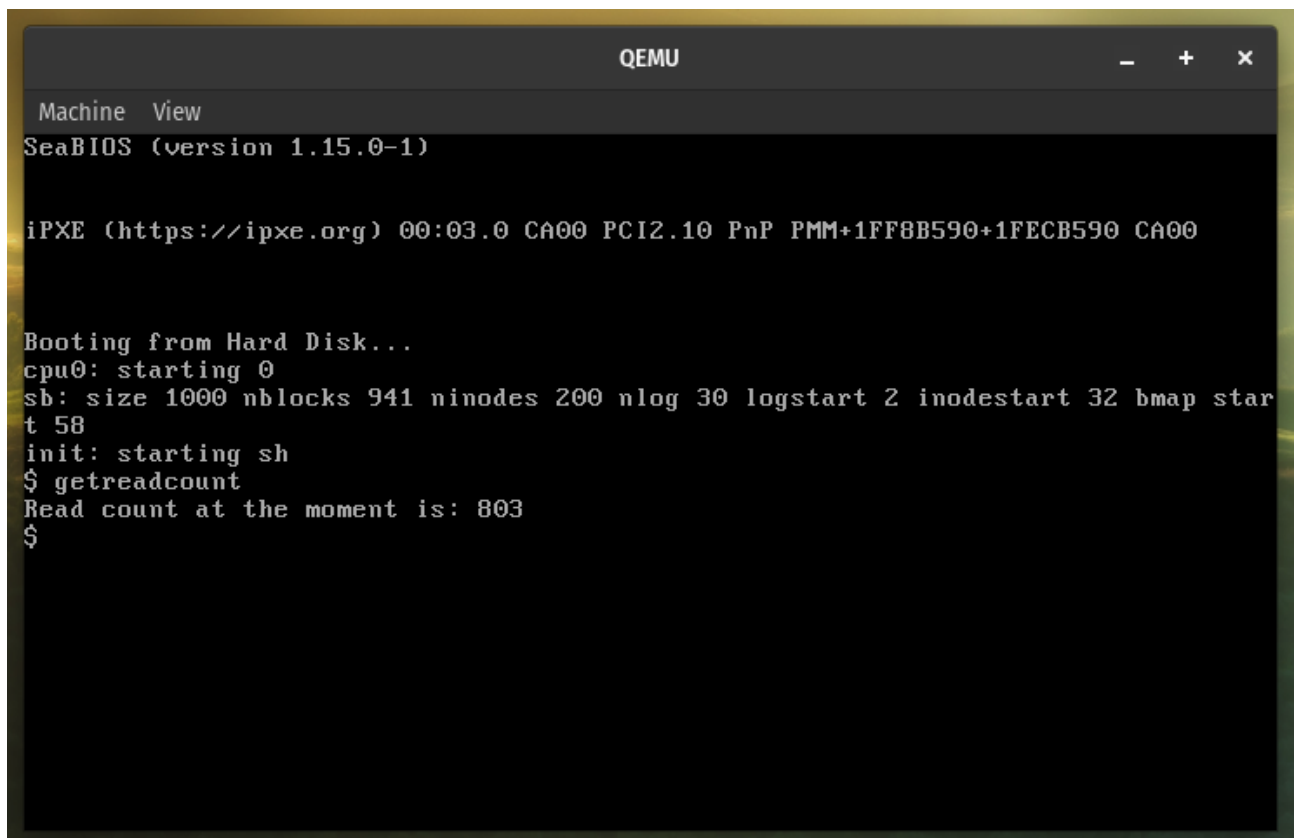
start the kernel by make and qemu.

```
otto@rivendell:~/Documents/Yliopisto/Käyttöjärjestelmä/CourseProject/xv6-public$ sudo make qemu
[sudo] password for otto:
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -c -o cat.o cat.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -c -o ulib.o ulib.c
gcc -m32 -gdwarf-2 -Wa,-divide -c -o usys.o usys.S
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -c -o printf.o printf.c
```

Should look like this:



Now you can run the getreadcount to get the readcount:



```
QEMU
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ getreadcount
Read count at the moment is: 803
$
```

You can close the qemu from the X-button.