Writers:
Shalev Ben-David
Almog Shor

Editor:
Dr. Nezer J. Zaidenberg

English Translation:
Roy Simanovich

Scribe (Course Summary)

Operating Systems

Table of Contents

Translator Notes

Important information

This is the scribe (course summary) of the course "Operating Systems" of the Computer Science department, at the Ariel University of Samaria, that was led by Dr. Nezer J. Zaidenberg, in Spring 2023. The original Hebrew version of this scribe was unordered, had lack of important information and had a lot of mistakes.

Warranty

There is no warranty on this version of the scribe. This version isn't approved by the professor and the translator doesn't take any warranty on mistakes that were made in this scribe version. Please use only the approved version of the scribe, that's located on the module website of this course.

Copyright notice

This version of the scribe is a direct translation of the official scribe for this course, which was written by three students and edited by the professor itself.

Addition work has been done by the translator itself for adding more information that's based on Linux's official manual, (Linux man pages) Beej's guides for networking (Hall) and interprocess communication (Hall), Linux Kernal Module Programming Guide (Salzman, Burian, Pomerantz, Mottram, & Huang, 2023), IBM's pthread library documentations (IBM, 2021), Pattern-Oriented Software Architecture Vol. 2 (Schmidt, Stal, Rohnert, & Buschmann, 2000), Wikipedia, and other sources that are available online.

Final thoughts

I worked too many hours for this version of the scribe, probably more than I should have, instead of learning for other courses. Please appreciate the work that was done here. One "thank you" will be appreciated. Thank you for reading, and good luck with the exams! Now I'll stop griding your gears, let's go over the material!

Introduction

Lecturer Name	Phone Number	Email Address	Reception time
Dr. Nezer J. Zaidenberg	054-553-1415	scipio@scipio.com	After the lecture Sunday/Thursday with appointment[1]

Few points about the course itself:

- The lectures would be recorded, without any guarantee, and there won't be any hybrid classes, so it's **recommended** to attend every lecture.
- Thursday's lectures will be taught through Zoom, no physical attendance.
- It's the professor's recommendation to pass this course, as the next year course would be changed drastically, and the next year course will be harder.
- In this course we won't write an operating system from scratch, but we will learn to work with Linux environment, write drivers, learn design patterns, etc.

Notes for the course's assignments:

- All the codes will be written in either C or C++ language.[2]
- Each week/two weeks a new assignment will be published.
- There will be about 7 assignments during the semester.[3]
- The assignments account for 40% of the final grade.

Notes about the final exam:

- 40 points for multiple choice questions.
- 60 points for open questions:
    - Finding a bug in a given code.
    - Question about designing a system (How would you write the code).
    - Writing a code (A valid C program).

About me:

I'm a new lecturer here at Ariel University. I have been teaching operating systems courses for more than 10 years, I've been a lecturer at the Open University, Tel-Aviv University, Bar-Ilan University, and more academic institutes. I adore the C programming language.

[1] To contact, please refer to WhatsApp, as emails can be missed.
[2] Except for kernel modules, which will be written in the C language only.
[3] In this specific semester there were 6 assignments (5 of the best are calculated to the final grade) and 2 bonus assignments.

Lecture 1[4]

UNIX

In 1970, the AT&T[5] company reached the conclusion that the activity of computers should go outside of the company. A company named "Bell Labs"[6] (owned by AT&T) that was responsible for developing important software including the C programming language, UNIX, etc.

In 1969, they wanted to buy the Multics[7] system, but didn't have the budget for it. They found a cheaper and more common system named PDP-11[8], and wrote an operating system that inherent some of the features of the Multics system, and called it UNIX[9]. They distributed the code of the OS they wrote, and it was developed over time in a way that several versions of UNIX came out. After that, more companies adapted the UNIX OS and developed their own OS that is based on UNIX principles.

To standardize the usage of UNIX custom distributions, Bell Labs created a test called POSIX[10] to make sure that the operating system withstands the standards and features of the original UNIX operating system.

Some operating systems like Mac OS (Classic), MS-DOS and IBM MVP (Mainframe computers) didn't budge to try imitating UNIX. Other that were based of UNIX didn't pass the POSIX test – those operating systems are called "UNIX-Like".

In the 80's, AT&T started to sue companies that used the UNIX name under their operating system, which didn't pass the POSIX test.

[4] In this lecture there were background stories and definitions that are essential to the final exam.
[5] An American worldwide communication corporation.
[6] A sub-division of AT&T that had a monopoly over the communication market in the USA. In 1984, the US supreme court ordered the company to disband to 8 different companies.
[7] A superior computer system that was considered a "top" (supercomputer) at that time, and was able to support multiple users, multiple processes, and multiple CPUs.
[8] A series of minicomputers that was a cheap alternative for mainframes and analog computers.
[9] A standard for operating systems that pass the POSIX test.
[10] A supplementary of technical standards that provides a basic functionally for operating systems.

Linux

In 1991, there was a poor Finn student named Linus Torvalds.[11] He learned computer science and wanted to install a UNIX machine in his home. In the university where Torvalds's learned, they teach by using UNIX-Like machines and used Intel's i386 chip (and Intel's i387 chip for floating point arithmetic).[12] In the past, those hardware components were bought by the professors themselves with huge budget.

Since Torvalds didn't have the budget to buy those components, he decided to build his own UNIX based operating system from scratch and call it "Linux"[13]. In August 1991 he published the source code of his creation and advised people to use it as a social network (Unset).

Because Torvalds was Finn and not American, the AT&T company didn't try to sue him, so he continued to distribute the source code to other people in the world. The project grew large, and many people started to contribute to it, and thus AT&T never stood a chance to try and sue anyone from using Linux.

Apple[14]

Apple had their own operating system that worked better than others. But users wanted to be allowed to multitasking, and thus Apple needed a new operating system that's based on UNIX but couldn't determine what to do. They planned to switch their operating system, and thus they bought NeXT[15] and used their operating system called "NeXTSTEP"[16], which was based of UNIX.

In 1999, Apple merge NeXTSTEP with their own vision, and thus Mac OS X was born, which later became macOS.

Relationship between Apple and Linux: Apple's architecture is different from Linux, but the usage of the system programming is practically identical.

---

[11] For general information: The three most popular Finn people are Swedish (In fact, 10% of Finn people are Swedish).
[12] Today those chips are obsolete.
[13] An operating system that's based on UNIX. The main difference between Linux and other UNIX operating systems is that it's free and open source. Linux is available and implemented on almost all devices across the world: mobile phones, security cameras, multimedia devices, routers, vehicles and even 90% of the TOP 500 list of supercomputers run Linux. Linux is even found on PC (mainly in Apple's version), and almost anywhere else.
[14] For self-reading.
[15] American computer company that founded by Steve Jobs back in 1985. Disbanded when Apple bought it in 1996.
[16] An operating system that is based on UNIX, is object oriented and supports multitasking.

License for free software[17]

In the 80's, at the MIT[18] university, there was a big and central printer. Richard Stallman added a feature to their printer to support printing titles to pages. At someday, Xerox[19], the printer's manufacture, decided to do a software upgrade to the printer and Stallman's feature was gone. Stallman asked Xerox to add his creation to the printer's official code, but they refused and claimed it will expose their proprietary code. They also didn't understand why he complained, as he got a free upgrade. Stallman disagreed, so he paid a lawyer to write him a free software license, also known as the "GNU General Public License" (or GNU GPL for short). By the license definition, a software will be considered free if the following 4 terms are applied:

- The software is free and could be run in any way and for any purpose.
- There is no warranty whatsoever.
- Anyone can receive a copy of the source code of the software, for a small payment.
- The only limitation for derived code[20] is to keep it also open source.

Linux distribution[21]

We want a comfortable Linux installation. There are two ways to install a Linux distribution:

- **Way 1** – Install using famous distributions like Red Hat, Ubuntu, etc. They provide comfortable tools to manage applications and packages and wrapped the installation to a more user-friendly interface.
- **Way 2** – Downloading and compiling each open-source library and the kernel manually, while porting them to be compliable with our hardware components. This is a long and painstaking process that repeats each time we need to upgrade a package. Another problem is that a third party can't check its own software on our custom distribution. For example, if I'm Oracle company and I want that my product will be able to run on client's computer without a problem, I need to install it on an identical system, but if the client's operating system contains different software pieces from different locations that he compiled manually, I can't clone his machine.

In this course we won't learn about Linux management and won't discuss about the difference between each distribution, we will only deal with system programming.[22]

---

[17] The summary is not accurate and is intended for Computer Science students and not as a substitute for a legal opinion.
[18] Massachusetts Institute of Technology (Private university in the US).
[19] An American company that creates and sells printers.
[20] A piece of code that was partly or fully taken from an existing piece of code (A creation that's based on another creation). According to GPL, a library that was used for a piece of code will make it a derived work. There is another license called LGPL (Lesser GNU), which allow using open-source libraries but keeping the code proprietary. In every industrial work and each time, you code with open-source libraries, it's recommended to request an opinion from the organization's legal department.
[21] An operating system that contains the core (kernel) of the Linux operating system, libraries, different files, and a workspace for operating the computer. Usually is open-sourced and free.
[22] Each assignment must be able to compile and run on Ubuntu 20 or Ubuntu 22. You can code it on whatever environment you want.

Lecture 2

Library functions

A library function is a function that's originated from a shared library (standard C library, custom libraries, etc.).

For example: $printf(3)$, $strcmp(3)$.

Advantage: Write once, accessible to all.

Importance: If we can't implement a function (for example, in the crypto world there is such assumption), or when we want to save time in programming and debugging.

Question: Why is it good that everyone uses the same encryption/shared library?
Answer: Sometimes programmers write bugs accidently, and as a result in the final product there will be some bugs, that the users need to fix themselves. In contrast, in a library, you can change the code once and it will affect any program that uses this function, without needing to rewrite the function for each program and recompiling it.[23]

Question: Is the $send(2)$ function that handles sockets, a library function?
Answer: **No**, a library function is a function that someone else wrote for you, but you can implement it yourself. In the $send(2)$ function, the function calls the operating system itself to execute a specific service, because the programmer doesn't have the privileges to access the network card directly, and as a matter of fact, he doesn't have the privileges to access any hardware component directly. Because the $send(2)$ function asks for a service that isn't controlled by the user, but by the operating system, it can't be a library function.

Question: Why?
Answer: In case there isn't an administrator (supervisor), a memory corruption could occur.

Question: How we will coordinate?
Answer: We'll configure those library functions as an administrator. An administrator in a computer is the operating system itself, and each time we want to access hardware or a process, we'll ask the given supervisor.

[23] Example of exceptional case: The heartbeat bug in the OpenSSL library that almost crashed the entire world, as everyone are using the OpenSSL library. Was discovered in 2014 and patched. Before that, nobody knew about any bug in the library itself for 17 years from its release to the public.

Man (Manual)

The man (short for manual) is the official Linux commands and functions manual. The man is divided into 8 different chapters. In this course we'll discuss only the first three:

- **Part 1 – man(1):** This chapter contains all the command line commands that could be run via a shell (for example, bash). Examples: $man(1)$, $ls(1)$.
- **Part 2 – man(2):** This chapter contains all the system calls[24] functions. For example: $send(2)$, $recv(2)$.
- **Part 3 – man(3):** This chapter contains most of the library functions, usually from the standard C library. Examples: $printf(3)$, $strcmp(3)$.

Question: What about weird entries like $printf(1)$ in man?
Answer: This is an historical command line tool, it isn't used anymore, but still present for backwards compatibility. Some entries from different chapters have the same name but it doesn't mean it describes the same functionality.[25]

The default of the man is to show the first chapter that this entry appears. If the wrong chapter is selected, you can change the viewed chapter by typing explicitly the chapter number. For example:

$$man\ 3\ printf$$
$$man\ 2\ kill$$

Will show the correct man pages for the $printf(3)$ and the $kill(2)$ entries.

Supervisor vs. Hypervisor: Processes shouldn't communicate with hardware directly. Suppose we have software like virtual machines, that allows two operating systems to communicate with the machine itself, so we want to manage the processes in an efficient way. We use a hypervisor for this. A hypervisor is an operating system that manages other operating systems, and it's above the supervisor, hence its name. In this course we won't touch on this subject.

Question: How does system call works?
Answer: The parameter we provide to the function is actually a number that we enter to the $SYSCALL$ and thus the operating system knowns which service we need, no matter what language we use. So, for example, the $send(2)$ system call will work the same both in C and PASCAL.

Question: And what about Java and Python?
Answer: We have a bytecode or a Python script that we run. There is an interpreter that reads the instructions and converts them to machine code for $SYSCALL$ and the operating system, even though it doesn't look like this in the code itself. Eventually, there is only one interface to communicate with the hardware, and it is the operating system itself.

---

[24] A system call ($SYSCALL$) is a way of communicating with the operating system. This is a different assembly instruction then $CALL$ (which is used to enter a function). System call assembly instruction is entering a function but also escalate privileges to the level of privileges that the operating system has, and thus be able to communicate with the hardware directly.
[25] For example, the $kill(2)$ system call also has a command line tool version, $kill(1)$, that's run directly from the shell and not via a compiled C program.

Privilege levels

In Intel's x86-64 architecture there are two[26] levels of privileges that we use:

- **Level 0** – Represent the operating system.
- **Level 3** – Represent the user itself.

In ARM architecture there are also two[27] levels of privileges that we use:

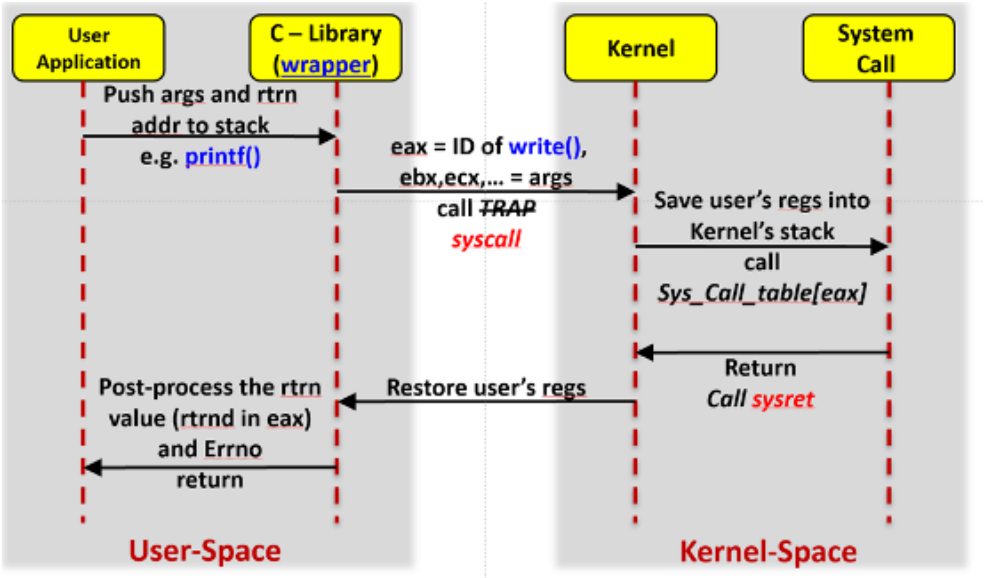
- **Privilege EL0** – User privilege.
- **Privilege EL1** – System administrator privilege.

We will use the term "to escalate privileges" (like in ARM architecture) to describe going up to operating system privileges, even though in Intel's architecture it's "going down".[28]

System calls can be roughly groped into five major categories:

- **Process control** (For example: create/terminate process)
- **File management** (For example: $read(2)$, $write(2)$)
- **Device management** (For example: logically attach a device – $mount(2)$)
- **Information maintenance** (For example: set time or date)
- **Communications** (For example: send messages – $send(2)$, $sendto(2)$)

Question: How it actually works?
Answer:



Combination: When we put on high priority the chapters 1, 2 and 3, we call the function by the location of it in the chapter.

[26] Level 1 is Hypervisor, which is out of the scope of this course.
[27] Except the hypervisor level.
[28] There is an interface to execute and manage CPU interrupts, but it's out of scope for this course.

Process management

$fork(2)$

This is the only function in C where we call it once and it returns twice – One time for the parent process and one time for the child process. The function returns with the process identifier (pid) of the child process to the parent process, and 0 to the child process. The fork system call clones the parent process and all its resources to the child process – sockets, files, and memory allocations.[29] [30]

From $man(2)$:

$fork()$ creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process. The child process and the parent process run in separate memory spaces. At the time of $fork()$ both memory spaces have the same content. Memory writes, file mappings ($mmap(2)$) and unmappings ($munmap(2)$) performed by one of the processes do not affect the other. The child process is an exact duplicate of the parent process except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group ($setpgid(2)$) or session.
- The child's parent process ID is the same as the parent's process ID.
- The child does not inherit its parent's memory locks ($mlock(2)$, $mlockall(2)$).
- Process resource utilizations ($getrusage(2)$) and CPU time counters ($times(2)$) are reset to zero in the child.
- The child's set of pending signals is initially empty ($sigpending(2)$).
- The child does not inherit semaphore adjustments from its parent ($semop(2)$).
- The child does not inherit process-associated record locks from its parent ($fcntl(2)$).[31]
- The child does not inherit timers from its parent ($setitimer(2)$, $alarm(2)$, $timer\_create(2)$).
- The child does not inherit outstanding asynchronous I/O operations from its parent ($aio\_read(3)$, $aio\_write(3)$), nor does it inherit any asynchronous I/O contexts from its parent (see $io\_setup(2)$).

Question: What's the difference between a process and a thread?
Answer: If we have a thread, everything is shared[32], and nothing is cloned. But if I have a process, everything is separated: Interruption table, files, sockets, etc.

In the UNIX standard we have only process and a thread. In Linux there is the $clone(2)$ system call which is only present in Linux. In this course we go by the UNIX standard, so we won't use it.[33]

---

[29] When the parent process has more than one thread, it only clones the thread that called the $fork(2)$ system call.
[30] It also important to notice that after a process is forked, everything is separated from now, meaning the child won't affect the parent and the parent won't affect the child.
[31] On the other hand, it does inherit $fcntl(2)$ open file description locks and flock(2) locks from its parent.
[32] Not accurate, we'll discusses it in the following lectures.
[33] The $clone(2)$ system call allows to control which resources a shared and which are separated.

Fork features:

- **CopyOnWrite** – The process of taking the entire memory of existing process and copying it completely to a different process is very expensive in resources. To optimize the process, we use the CopyOnWrite (or CoW for short) of the $fork(2)$ system call to only copy the pointers of the memory pages that the parent process has. If the child process changes the page, it only copies that page and changes it accordingly.
- $\boldsymbol{flush}()$ – This function flushes the buffer of the operating system.

Question: In C++, does writing to $cout$ writes to the buffer?
Answer: Not necessarily. C++ changes every period. In the earlier standards of C++, $cout$ worked without a buffer. In later revisions, $cout$ is now a buffered output.[34]

Question: What's the first line of code that's executed when we start a program?
Answer: The code that's executed before the $main()$ function is called $start$ and it's a special piece of code ($CTOR$), that calls via the operating system to the $main()$ function.

$\boldsymbol{wait(2)}$

A parent process can get the returned value of the child process. The parent process can also get information about how the child process end running (signal, core dumped, or normal termination). To make the parent process wait for this information, we can use the wait(2) system call.

From $\boldsymbol{man(2)}$:

All of these system calls are used to wait for state changes in a child of the calling process and obtain information about the child whose state has changed. A state change is considered to be the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state. If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call.[35] In the remainder of this page, a child whose state has changed, and which has not yet been waited upon by one of these system calls is termed waitable.

A zombie process is not an actual process, but was a child process in its lifetime, and ended in some way (either by crash or normal exit). When it ended, the process needed to send its ending state to the parent process (via $wait(2)$). If the parent process already finished before the child process got a chance to return the values, the operating system will take control and the "zombie" process will be adapted via the INIT[36] process. This process doesn't do anything, it just waits for all his child's and kills them instantly, and thus the zombie process died and all the resources that it was holding were released.

[34] It's not a C++ course so we won't go deeply into the standards.
[35] Assuming that system calls are not automatically restarted using the $SA\_RESTART$ flag of $sigaction(2)$.
[36] INIT is process number 1 (Process ID = 1), the common ancestor for all the processes in UNIX.

We have three different types of wait functions, from different versions of different standards.

$execl(2)$, $execlp(2)$, $execle(2)$, $execv(2)$, $execvp(2)$ and $execvpe(2)$

So far, we know how to clone processes, but how can we run a completely new process? To replace the existing process, we can use the $execve(2)$ system call and the exec system call family. There are 6 system calls in this family each one with different parameters and in the end, each one calls the $execve(2)$ system call. When we call one of the exec system calls after a $fork(2)$ system call, it takes care that the operating system will release all the resources that the existing process it takes and assigning new resources to the new process we want to run.

<u>From $man(2)$</u>:

$execve()$ executes the program referred to by pathname. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

- $pathname$ must be either a binary executable, or a script. $argv$ is an array of pointers to strings passed to the new program as its command-line arguments. By convention, the first of these strings (i.e., $argv[0]$) should contain the filename associated with the file being executed.
- The $argv$ array must be terminated by a NULL pointer.[37]

$execve()$ does not return on success, and the text, initialized data, uninitialized data (bss), and stack of the calling process are overwritten according to the contents of the newly loaded program. If the current program is being ptraced, a SIGTRAP signal is sent to it after a successful $execve()$.

All process attributes are preserved during an $execve()$, except the following:

- Dispositions of any signals that are being caught are reset to the default ($signal(7)$).
- Any alternate signal stack is not preserved ($sigaltstack(2)$).
- Memory mappings are not preserved ($mmap(2)$).
- Attached System V shared memory segments are detached ($shmat(2)$).
- POSIX shared memory regions are unmapped ($shm\_open(3)$).
- POSIX timers are not preserved ($timer\_create(2)$).
- Any open directory streams are closed ($opendir(3)$).
- Memory locks are not preserved ($mlock(2)$, $mlockall(2)$).
- Exit handlers are not preserved ($atexit(3)$, $on\_exit(3)$).

<u>Question</u>: Why does it work like this?
<u>Answer</u>: No particular reason, but its efficient as we use the CoW feature to effectivity manage our resources.

<u>Question</u>: What's a $SYSTEM$?
<u>Answer</u>: A SYSTEM is a library that runs those 3 systems calls one after other: $fork(2)$, $exec(2)$, $wait(2)$.

[37] Thus, in the new program, $argv[argc]$ will be NULL.

Error handling

When we want to print an error message of the error that occurred, we can user the $perror(3)$ library function, that uses the global variable called $errno$ (which determines what error occurred).[38] When we call this function, a corresponding error will be printed to the stderr file descriptor.[39]

From $man(3)$:

The $perror(const\ char\ *s)$ function produces a message on standard error describing the last error encountered during a call to a system or library function. First, the argument string s is printed, followed by a colon and a blank.[40] Then an error message corresponding to the current value of $errno$ and a new line. To be of most use, the argument string should include the name of the function that incurred the error. The global error list $sys\_errlist[]$, which can be indexed by $errno$, can be used to obtain the error message without the newline. The largest message number provided in the table is $sys\_nerr - 1$. Be careful when directly accessing this list because new error values may not have been added to $sys\_errlist[]$. The use of $sys\_errlist[]$ is nowadays deprecated; use $strerror(3)$ instead. When a system call fails, it usually returns -1 and sets the variable $errno$ to a value describing what went wrong.[41] Many library functions do likewise. The function $perror()$ serves to translate this error code into human-readable form. Note that $errno$ is undefined after a successful system call or library function call: this call may well change this variable, even though it succeeds, for example because it internally used some other library function that failed. Thus, if a failing call is not immediately followed by a call to $perror()$, the value of errno should be saved.

[38] **That is not accurate:** The variable errno is thread specific storage, which is out of scope for this course.
[39] In Windows there is a similar solution with a library function that's called $getlasterror()$.
[40] if s is not NULL and *s is not a null byte ('\0').
[41] These values can be found in $< errno.h >$ header file.

Lecture 3

File descriptors

A file descriptor can describe many things like files, directories, sockets, etc.[42] For each process, the operating system holds a table of all the files that were opened by that process. Each file in the table is indexed by a non-negative incrementing number (key). We call this index a file descriptor. Whenever a process is created, 3 file descriptors are opened automatically: $stdin$ (0), $stdout$ (1) and $stderr$ (2). Details of those file descriptors are in following table:

FD	Name	Description
0	Standard Input ($stdin$)	The standard input of the machine (usually a keyboard but can be redirected to be a file input). End of input is represented by the EOF (End-of-file) special character.
1	Standard Output ($stdout$)	The standard output of the machine (usually a screen or a terminal but can be redirected to be a file output). The output is buffered and won't flush without the "\n" character or manually flushing the buffer with $fflush(stdout)$ library function.
2	Standard Error ($stderr$)	That standard error output of the machine (usually a screen or a terminal but can be redirected to be a file output). The output is unbuffered.

When we call $fork(2)$, all the file descriptors of the parent process are cloned to the child process.[43]

We can also use the $dup(2)$ system call to deep copy one file descriptor to a new one, which will be allocated as the lowest file descriptor index available. The $dup2(2)$ system call does the same, but we can control the file descriptor index.

<u>From $man(2)$</u>:

The $dup(int\ oldfd)$ system call allocates a new file descriptor that refersto the same open file description as the descriptor $oldfd$.[44] The new file descriptor number is guaranteed to be the lowest-numbered file descriptor that was unused in the calling process. After a successful return, the old and new file descriptors may be used interchangeably. Since the two file descriptors refer to the same open file description, they share file offset and file status flags; for example, if the file offset is modified by using $lseek(2)$ on one of the file descriptors, the offset is also changed for the other file descriptor. The two file descriptors do not share file descriptor flags (the close-on-exec flag). The close-on-exec flag (FD_CLOEXEC; see $fcntl(2)$) for the duplicate descriptor is off.

[42] In the original design of UNIX (from the 70's), everything is based on inheritance from a process or a file. In the 90's, Linux added a virtual file system that's called $/prog$ and contain virtual files that each one represents a process that's running on the machine, extending the concept that "everything" is a file.

[43] Please note that the cloning is done in a deep copy manner: There are two different tables of file descriptors, one is the original of the parent process and the other is the cloned one of the child processes. If the parent or the child processes open another file descriptor, it won't affect the other.

[44] For an explanation of open file descriptions, see $open(2)$.

The $dup2(int\ oldfd, int\ newfd)$ system call performs the same task as $dup()$, but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in $newfd$. In other words, the file descriptor $newfd$ is adjusted so that it now refers to the same open file description as $oldfd$. If the file descriptor $newfd$ was previously open, it is closed before being reused; the close is performed silently.[45] The steps of closing and reusing the file descriptor $newfd$ are performed atomically. This is important, because trying to implement equivalent functionality using $close(2)$ and $dup()$ would be subject to race conditions, whereby $newfd$ might be reused between the two steps. Such reuse could happen because the main program is interrupted by a signal handler that allocates a file descriptor, or because a parallel thread allocates a file descriptor.

Note the following points:

- If $oldfd$ is not a valid file descriptor, then the call fails, and $newfd$ is not closed.
- If $oldfd$ is a valid file descriptor, and $newfd$ has the same value as $oldfd$, then $dup2()$ does nothing and returns $newfd$.

Signals

Signals are a way for UNIX to implement interrupts - messages from the operating system itself to processes.[46]

There are two types of signals:

- **Synchronous** – Related to logical errors in the code itself, for ex. dividing by zero, accessing invalid/not permitted memory address, illegal assembly instructions.
- **Asynchronous** – Result of things that aren't related to the code, and are more unpredictable events like request for the process to terminal, external event, etc.

Signal list in UNIX:

No.	Name	Description
1	SIGHUP	If a process is being run from terminal and that terminal suddenly goes away, then the process receives this signal.
2	SIGINT	The process was "interrupted". This happens when you press Control+C on the controlling terminal.
3	SIGQUIT	The SIGQUIT signal is similar to SIGINT, except that it's controlled by a different key—the QUIT character, usually C-\—and produces a core dump when it terminates the process, just like a program error signal. You can think of this as a program error condition "detected" by the user.
4	SIGILL	Illegal instruction. The program contained some machine code the CPU can't understand.
5	SIGTRAP	This signal is used mainly from within debuggers and program tracers.
6	SIGABRT	The program called the $abort(3)$ function. This is an emergency stop.

[45] For example, any errors during the close are not reported by $dup2()$.
[46] Usually because of an illegal instruction.

No.	Name	Description
7	SIGBUS	An attempt was made to access memory incorrectly. This can be caused by alignment errors in memory access etc.
8	SIGFPE	A floating-point exception happened in the program.
9	SIGKILL	The process was explicitly killed by somebody wielding the $kill(1)$ program. Kills immediately and can't be ignored.
10	SIGUSR1	Left for the programmers to do whatever they want.
11	SIGSEGV	An attempt was made to access memory not allocated to the process. This is often caused by reading off the end of arrays etc.
12	SIGUSR2	Left for the programmers to do whatever they want.
13	SIGPIPE	If a process is producing output that is being fed into another process that consume it via a pipe ("$producer \mid consumer$") and the consumer dies, then the producer is sent this signal.
14	SIGALRM	A process can request a "wake up call" from the operating system at some time in the future by calling the $alarm()$ function. When that time comes round the wake-up call consists of this signal.
15	SIGTERM	The process was explicitly killed by somebody wielding the $kill(1)$ program.
16	–	Unused.
17	SIGCHLD	The process had previously created one or more child processes with the $fork(2)$ function. One or more of these processes has since died.
18	SIGCONT	If a process has been paused by sending it SIGSTOP then sending SIGCONT to the process wakes it up again ("continues" it). Used by debuggers.
19	SIGSTOP	If a process is sent SIGSTOP it is paused by the operating system. All its state is preserved ready for it to be restarted (by SIGCONT) but it doesn't get any more CPU cycles until then. Used by debuggers like GDB (GNU Debugger).
20	SIGTSTP	Essentially the same as SIGSTOP. This is the signal sent when the user hits Control+Z on the terminal. (SIGTSTP is short for "terminal stop") The only difference between SIGTSTP and SIGSTOP is that pausing is only the default action for SIGTSTP but is the required action for SIGSTOP. The process can opt to handle SIGTSTP differently but gets no choice regarding SIGSTOP.
21	SIGTTIN	The operating system sends this signal to a backgrounded process when it tries to read input from its terminal. The typical response is to pause (as per SIGSTOP and SIFTSTP) and wait for the SIGCONT that arrives when the process is brought back to the foreground.
22	SIGTTOU	The operating system sends this signal to a backgrounded process when it tries to write output to its terminal. The typical response is as per SIGTTIN.
23	SIGURG	The operating system sends this signal to a process using a network connection when "urgent" out of band data is sent to it.
24	SIGXCPU	The operating system sends this signal to a process that has exceeded its CPU limit.
25	SIGXFSZ	The operating system sends this signal to a process that has tried to create a file above the file size limit.
26	SIGVTALRM	This is very similar to SIGALRM, but while SIGALRM is sent after a certain amount of real time has passed, SIGVTALRM is sent after a certain amount of time has been spent running the process.

No.	Name	Description
27	SIGPROF	This is also very similar to SIGALRM and SIGVTALRM, but while SIGALRM is sent after a certain amount of real time has passed, SIGPROF is sent after a certain amount of time has been spent running the process and running system code on behalf of the process.
28	SIGWINCH[47]	A process used to be sent this signal when one of its windows was resized.
29	SIGIO	process can arrange to have this signal sent to it when there is some input ready for it to process or an output channel has become ready for writing.
30	SIGPWR	A signal sent to processes by a power management service to indicate that power has switched to a short-term emergency power supply. The process (especially long-running daemons) may care to shut down cleanlt before the emergency power fails.
31	SIGSYS	Unused.

Difference between interrupts and signals:

- **Interrupt** – A term from the hardware world[48] that the kernel handles.
- **Signal** – Software interrupt, a message from the operating system itself to a specific process. A process can catch and handle this signal accordingly.[49]

Signal handling

There are 5 different possible default actions for handling signals:

Action	Description
Exit	Forces the process to be terminated.
Core	Forces the process to be terminated and creates a core dump file for debug purposes.
Stop	Stops the process.
Continue	Continue a stopped process.
Ignore	Ignore the signal – do nothing.

There are two signals that can't be caught and handled, and the default action will always take place: $SIGKILL$ (Terminate) and $SIGSTOP$ (Stop).

To catch signals, we can use on of the following system calls: $signal(2)$ or $sigaction(2)$, where the second one is newer and has a defined behavior according to the standard. The signal handling function receives an integer that represents the signal number and returns nothing (void). We can also use the $sigprocmask(2)$ system call to mask signals.

[47] Mostly unused these days.
[48] For example, a packet from the network has been detected by the network interface device, or a replay for a DMA request from the disk.
[49] For example, by closing files and any active sockets.

From $man(2)$:

$signal(int\ signum, sighandler\_t\ handler)$ sets the disposition of the signal signum to handler, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal handler"). If the signal signum is delivered to the process, then one of the following happens:

- If the disposition is set to SIG_IGN, then the signal is ignored.
- If the disposition is set to SIG_DFL, then the default action associated with the signal occurs.
- If the disposition is set to a function, then first either the disposition is reset to SIG_DFL, or the signal is blocked, and then handler is called with argument signum. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.

The $sigaction(int\ signum, const\ struct\ sigaction\ *restrict\ act, struct\ sigaction\ *restrict\ oldact)$ system call is used to change the action taken by a process on receipt of a specific signal. $signum$ specifies the signal and can be any valid signal except $SIGKILL$ and $SIGSTOP$.

$sigprocmask(int\ how, const\ sigset\_t\ *restrict\ set, sigset\_t\ *restrict\ oldset)$ is used to fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. The behavior of the call is dependent on the value of how, as follows:

- SIG_BLOCK – The set of blocked signals is the union of the current set and the set argument.
- $SIG\_UNBLOCK$ – The signals in set are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- $SIG\_SETMASK$ – The set of blocked signals is set to the argument set.

If $oldset$ is non-NULL, the previous value of the signal mask is stored in $oldset$. If set is NULL, then the signal mask is unchanged (i.e., how is ignored), but the current value of the signal mask is nevertheless returned in $oldset$ (if it is not NULL). A set of functions for modifying and inspecting variables of type $sigset\_t$ ("signal sets") is described in $sigsetops(3)$.

Signals in real time systems:

In real time system the behavior is predictable. For example, if we have an urgent task (like stabilizing a jet fighter) and we have a non-urgent task, the urgent task will always get CPU time as it's at higher priority. In a workstation, that's not the behavior we want, as if a task is stuck (for example, in an infinite loop), if it has high priority, it will consume all the CPU time and will leave no room for other processes. There are ways to handle signals in real time system.[50]

[50] This material is out of the scope of the course. Other material that is out of scope: Security aspect - system administrator can send signals to all the processes in the machine, any other user can send signals only to the processes he created.

Process group ID

Process group is a group of processes that share a common parent. Each process group is assigned an ID. We can send a signal to a process group, meaning we can send signals in one go to all the processes that are in the same group.

There are some system call functions that are related to process group id:

- $getpid(2)$ – Returns the process id of the current process.
- $getppid(2)$ – Returns the process id of the parent process of the current process.
- $getpgrp(2)$ – Returns the process group id that the current process is a part of.
- $setpgrp(2)$ – Sets the process group id of a given process.

From $man(2)$:

$getpid()$ returns the process ID (PID) of the calling process.[51]

$getppid()$ returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using $fork(2)$, or, if that process has already terminated, the ID of the process to which this process has been reparented.[52]

$getpgid()$ returns the $PGID$ of the process specified by pid. If pid is zero, the process ID of the calling process is used.[53]

$setpgid()$ sets the $PGID$ of the process specified by pid to $pgid$. If pid is zero, then the process ID of the calling process is used. If $pgid$ is zero, then the $PGID$ of the process specified by pid is made the same as its process ID. If $setpgid()$ is used to move a process from one process group to another[54], both process groups must be part of the same session.[55] In this case, the $pgid$ specifies an existing process group to be joined and the session ID of that group must match the session ID of the joining process.

[51] This is often used by routines that generate unique temporary filenames.
[52] Either $init(1)$ or a "subreaper" process defined via the $prctl(2)$ $PR\_SET\_CHILD\_SUBREAPER$ operation.
[53] Retrieving the $PGID$ of a process other than the caller is rarely necessary, and the POSIX.1 $getpgrp()$ is preferred for that task.
[54] As is done by some shells when creating pipelines.
[55] See $setsid(2)$ and $credentials(7)$.

Lecture 4

Computer networking

Definitions and common terms

- **Communication** – A series of APIs that allow information exchange between two processes using sockets.
- **Socket** – A half-duplex communication channel. Whenever we communicate with another entity, there is a socket.
- **Design patterns** – General, reusable solution to a commonly occurring problem within a given context in software design.
- **Factory pattern** – A common design pattern that uses mapping (hashing) between a key to a relevant constructor. In software engineering, the Factory Method pattern is a design pattern that creates different object that share a common and uniform interface, without knowing their inner classes implementation. The use of this design pattern is to configure an interface for creating objects, while allowing the sub-classes that are implemented to decide which object to create.

  Examples:
  - A common example of this design pattern is codec for video files. When we download videos, each video is encoded with some kind of codec. Whenever we play a video file in a program like Media Player Classic or VLC, those programs aren't related to any of the codecs, but when the program receives the video metadata, it creates a codec via the Factory method, that's based on the codec that was used to encode the video, to decode the video back to something playable.
  - Another example is a graphic (game) engine, that uses the Factory method to create objects: Characters, items, etc.[56]
  - In the computer communication realm, sockets use the Factory method by receiving 3 parameters and creating a socket that is based on those paraments.

[56] There are also companies that are specialize in building those graphic engines, like Unity.

The Internet Protocol (IP)

The Internet Protocol (IP) is a protocol used for communication between devices on the internet. It is a fundamental protocol that enables the exchange of information between different computers and networks. The IP protocol is responsible for ensuring that data packets are transmitted across the internet in a reliable and efficient manner. IP operates at the network layer of the TCP/IP protocol suite, which is a collection of protocols used for communication between devices on the internet. It is responsible for addressing and routing packets of data across the internet. The IP protocol is designed to be independent of the underlying physical network and can work with a variety of network technologies. The IP protocol is a connectionless protocol, which means that it does not establish a dedicated connection between the sender and receiver of data. Instead, data packets are transmitted independently and may take different routes to reach their destination. The IP protocol is designed to handle packet loss and reordering, and it uses a range of techniques to ensure that data is transmitted reliably.

One of the key features of the IP protocol is its ability to fragment and reassemble data packets. If a data packet is too large to be transmitted across a particular network segment, the IP protocol can break the packet into smaller pieces, or fragments, which can be transmitted separately. The receiving device can then reassemble the fragments into the original packet. This fragmentation and reassembly process is transparent to applications and higher-level protocols, which can treat the transmitted data as if it were a single, contiguous block. There are two major versions of the Internet Protocol: **IPv4** and **IPv6**. In addition to these two versions, there have been several other IP protocol versions that were developed but never widely adopted or released to the public.

- **IPv4** – The first version to be released to the public, and still in wide use to this day. The IPv4 addresses are 32-bit in length, meaning that there are 2^{32} theoretical addresses, but due to technical limitations and standardization, the amount of actual available addresses is much less.[57] In the past, people thought that 4 billion is astronomical number of devices and nobody imagined that we'll reach this limitation. Another contributing factor to the limitation is that large organizations got huge addresses space for themselves[58], and even the United States Department of Defense reserved for itself millions of addresses[59], meaning there were huge blocks of addresses that were gone to waste. Due to those limitations, there aren't enough IP addresses for every device on the world. An IPv4 address looks something like this: $X.Y.Z.W$. For example: $192.0.2.111$.

[57] Class A private networks ($0.0.0.0/8$, $10.0.0.0/8$, $100.64.0.0/10$ and $127.0.0.0/8$ blocks) take 134,525,952 addresses. Class B private networks ($169.254.0.0/16$ and $172.16.0.0/12$ blocks) take 1,114,112 addresses. Class C reserved addresses ($192.0.0.0/29$, $192.0.0.8/32$, $192.0.0.9/32$, $192.0.0.10/32$, $192.0.0.170/31$, $192.0.0.11 - 169$, $192.0.0.172 - 255$, $192.31.196.0/24$, $192.52.193.0/24$, $192.168.0.0/16$, $192.175.48.0/24$ and $198.18.0.0/15$ blocks) take 197,630 addresses. Classes D (multicast) & E (reserved) take 536,870,912 addresses. Total actual available addresses: 3,622,258,690 addresses.

[58] Companies like Xerox, MIT, HP, IBM, Prudential Securities (block $48.0.0.0/8$), Mercedes-Benz (block $53.0.0.0/8$), Ford (block $19.0.0.0/8$), AT&T (block $12.0.0.0/8$) and Apple (block $17.0.0.0/8$).

[59] The United States Department of Defense have 234,881,024 addresses (Blocks $6.0.0.0/8$, $7.0.0.0/8$, $11.0.0.0/8$, $21.0.0.0/8$, $22.0.0.0/8$, $26.0.0.0/8$, $28.0.0.0/8$, $29.0.0.0/8$, $30.0.0.0/8$, $33.0.0.0/8$, $55.0.0.0/8$, $205.0.0.0/8$, $214.0.0.0/8$ and $215.0.0.0/8$) reserved.

- **IPv6** – This is the most recent version of the Internet Protocol and was designed to address the limitations of IPv4, which was running out of available IP addresses. IPv6 uses 128-bit addresses,[60] compared to the 32-bit addresses used by IPv4, and also includes other improvements such as better support for mobile devices and improved security features. The IPv6 address uses the upper 64 bits to identify the subnet itself (the user), and the lower 64 bits is used to identify the NAT itself. Example for IPv6 address: $2001{:}\,0db8{:}\,c9d2{:}\,aee5{:}\,73e3{:}\,934a{:}\,a5ae{:}\,9551$. The $::1$ IPv6 address is used for loopback.
  Features that were added to IPv6 include:[61]
  - **Checksum** – No checksum is generated duo to the fact that TCP and UDP protocols do checksums themselves.
  - **Jumbo Packets** – Protocol headers take space, so IPv6 allows jumbo packets to ignore the headers overhead.
  - **Multicast** – Implemented on IPv4, but never used practically. For situations where we want to broadcast the transmission to a lot of stations simultaneously and effectively.

It's worth noting that IPv4 remains in widespread use today, despite the availability of IPv6. This is due in part to the fact that many older devices and networks are not compatible with IPv6, and also because the transition to IPv6 is a complex process that requires significant infrastructure changes.

[60] Which allow theoretical maximum of 2^{128} IPv6 addresses.
[61] The IPv6 protocol includes more features, which are out of scope for this course.

Socket system calls

$socket(2)$

A Factory method system call that creates a socket with the given parameters.

Common socket types:

- **AF_INET** – IPv4 version, used with $SOCK\_STREAM$ (TCP), $SOCK\_DATAGRAM$ (UDP) and $SOCK\_RAW$ (RAW Socket).
- **AF_INET6** – IPv6 version, used with $SOCK\_STREAM$ (TCP), $SOCK\_DATAGRAM$ (UDP) and $SOCK\_RAW$ (RAW Socket).
- **AF_UNIX** – Unix Socket Domain (UDS). Used with $SOCK\_STREAM$ (Stream) and $SOCK\_DATAGRAM$ (Datagram).

From $man(2)$:

$int\ socket(int\ domain, int\ type, int\ protocol)$ creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process. The domain argument specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in $< sys/socket.h >$ header file. The formats currently understood by the Linux kernel include:

Protocol name	Description
AF_UNIX	Local communication[62]
AF_LOCAL	Synonym for AF_UNIX
AF_INET	IPv4 Internet protocols[63]
AF_AX25	Amateur radio AX.25 protocol
AF_IPX	IPX - Novell protocols
AF_APPLETALK	AppleTalk
AF_X25	ITU-T X.25 / ISO-8208 protocol
AF_INET6	IPv6 Internet protocols[64]
AF_DECnet	DECet protocol sockets
AF_KEY	Key management protocol, originally developed for usage with IPsec
AF_NETLINK	Kernel user interface device
AF_PACKET	Low-level packet interface
AF_RDS	Reliable Datagram Sockets (RDS) protocol
AF_PPPOX	Generic PPP transport layer, for setting up L2 tunnels (L2TP and PPPoE)
AF_LLC	Logical link control (IEEE 802.2 LLC) protocol
AF_IB	InfiniBand native addressing
AF_MPLS	Multiprotocol Label Switching
AF_CAN	Controller Area Network automotive bus protocol
AF_TIPC	TIPC, "cluster domain sockets" protocol
AF_BLUETOOTH	Bluetooth low-level socket protocol
AF_ALG	Interface to kernel crypto API
AF_VSOCK	VSOCK (originally "VMWare VSockets") protocol for hypervisor-guest communication.
AF_KCM	KCM (kernel connection multiplexer) interface
AF_XDP	XDP (express data path) interface

[62] Supports Stream, Datagram and RAW.
[63] Supports TCP, UDP and RAW.
[64] Supports TCP, UDP and RAW.

The socket has the indicated type, which specifies the communication semantics. Currently defined types are:[65]

- **SOCK_STREAM** – Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.
- **SOCK_DGRAM** – Supports datagrams.[66]
- **SOCK_SEQPACKET** – Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each input system call.
- **SOCK_RAW** – Provides raw network protocol access.
- **SOCK_RDM** – Provides a reliable datagram layer that does not guarantee ordering.
- **SOCK_PACKET** – Obsolete and should not be used in new programs.[67]

Since Linux 2.6.27, the $type$ argument serves a second purpose: in addition to specifying a socket type, it may include the bitwise OR of any of the following values, to modify the behavior of $socket(2)$:

- **SOCK_NONBLOCK** – Set the O_NONBLOCK file status flag on the open file description[68] referred to by the new file descriptor. Using this flag saves extra calls to $fcntl(2)$ to achieve the same result.
- **SOCK_CLOEXEC** – Set the close-on-exec ($FD\_CLOEXEC$) flag on the new file descriptor.[69]

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the "communication domain" in which communication is to take place.[70] [71]

Sockets of type **SOCK_STREAM** are full-duplex byte streams. They do not preserve record boundaries. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a $connect(2)$ call. Once connected, data may be transferred using $read(2)$ and $write(2)$ calls or some variant of the $send(2)$ and $recv(2)$ calls. When a session has been completed a $close(2)$ may be performed. Out-of-band data may also be transmitted as described in $send(2)$ and received as described in $recv(2)$.

[65] Some socket types may not be implemented by all protocol families.
[66] Connectionless, unreliable messages of a fixed maximum length.
[67] See $packet(7)$.
[68] See $open(2)$.
[69] See the description of the O_CLOEXEC flag in $open(2)$ for reasons why this may be useful.
[70] See $protocols(5)$.
[71] See $getprotoent(3)$ on how to map protocol name strings to protocol numbers.

$bind(2)$

Whenever we create a new socket, we need to bind it to a specific port in our machine, "to catch a port". The port number is managed and used by the kernel to link incoming packets to the corresponding application socket. The port numbers are usually fixed.[72]

The $addr$ parameter must match the current socket that is used.

The $sockaddr\_in$ struct, which is used with AF_INET (IPv4), should look like this:

```
struct sockaddr_in {
        short sin_family;
        unsigned short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

Arguments breakdown:

- sin_family – The family of the socket.[73]
- sin_port – The port of the socket.[74]
- sin_addr – The IPv4 address in binary form, the result of conversion using $inet\_pton(3)$.
- sin_zero – Should be a bunch of zeros.

The $sockaddr\_in6$ struct, which is used with AF_INET6 (IPv6), should look like this:

```
struct sockaddr_in6 {
        sa_family_t sin6_family;
        in_port_t sin6_port;
        uint32_t sin6_flowinfo;
        struct in6_addr sin6_addr;
        uint32_t sin6_scope_id;
};
```

Arguments breakdown:

- $sin6\_family$ – The family of the socket.[75]
- $sin6\_port$ – The port of the socket.[76]
- $sin6\_flowinfo$ – IPv6 flow information.[77]
- $sin6\_addr$ – The IPv6 address in binary form, the result of conversion using $inet\_pton(3)$.
- $sin6\_scope\_id$ – Scope ID.[78]

[72] For example, port 80 is the port number for web communication.
[73] Should be AF_INET.
[74] Use the $htons(3)$ function to convert port from host-endian to network-endian.
[75] Should be AF_INET6.
[76] Use the $htons(3)$ function to convert port from host-endian to network-endian.
[77] Out of scope for this course.
[78] Out of scope for this course.

The $sockaddr\_un$ struct, which is used with AF_UNIX (UDS), should look like this:

```
struct sockaddr_un {
        sa_family_t sun_family;
        char sun_path[108];
};
```

Arguments breakdown:

- sun_family – The family of the socket.[79]
- sun_path – Contains null-terminated pathname. A UNIX domain socket can be bound to a null-terminated filesystem pathname.

From $man(2)$:

When a socket is created with $socket(2)$, it exists in a name space (address family) but has no address assigned to it. $int\ bind(int\ sockfd, const\ struct\ sockaddr\ *\ addr, socklen\_t\ addrlen)$ assigns the address specified by $addr$ to the socket referred to by the file descriptor $sockfd$. $addrlen$ specifies the size, in bytes, of the address structure pointed to by $addr$. Traditionally, this operation is called "assigning a name to a socket".

It is normally necessary to assign a local address using $bind()$ before a $SOCK\_STREAM$ socket may receive connections.[80] The rules used in name binding vary between address families.[81] The actual structure passed for the $addr$ argument will depend on the address family.[82]

$listen(2)$

The $listen(2)$ system call doesn't do anything in the mean of communication but only memory allocation (malloc) for stream sockets only.[83] If we won't allocate memory, any incoming connection will fail.

From $man(2)$:

$int\ listen(int\ sockfd, int\ backlog)$ marks the socket referred to by $sockfd$ as a passive socket, that is, as a socket that will be used to accept incoming connection requests using $accept(2)$. The $sockfd$ argument is a file descriptor that refers to a socket of type $SOCK\_STREAM$ or $SOCK\_SEQPACKET$. The backlog argument defines the maximum length to which the queue of pending connections for $sockfd$ may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of $ECONNREFUSED$ or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.

---

[79] Should be AF_UNIX.
[80] See $accept(2)$.
[81] Consult the manual entries in Section 7 for detailed information. For AF_INET, see $ip(7)$; for AF_INET6, see $ipv6(7)$; for AF_UNIX, see $unix(7)$; for $AF\_APPLETALK$, see $ddp(7)$; for AF_PACKET, see $packet(7)$; for AF_X25, see $x25(7)$; and for $AF\_NETLINK$, see $netlink(7)$.
[82] The only purpose of the $sockaddr$ structure is to cast the structure pointer passed in $addr$ in order to avoid compiler warnings.
[83] Trying to call $listen(2)$ on datagram sockets will result an undefined behavior.

To accept connections, the following steps are performed:

1. A socket is created with $socket(2)$.
2. The socket is bound to a local address using bind(2), so that other sockets may be $connect(2)ed$ to it.
3. A willingness to accept incoming connections and a queue limit for incoming connections are specified with $listen()$.
4. Connections are accepted with $accept(2)$.

The behavior of the backlog argument on TCP sockets changed with Linux 2.2. Now it specifies the queue length for completely established sockets waiting to be accepted, instead of the number of incomplete connection requests. The maximum length of the queue for incomplete sockets can be set using $/proc/sys/net/ipv4/tcp\_max\_syn\_backlog$. When $syncookies$ are enabled, there is no logical maximum length and this setting is ignored.[84] If the backlog argument is greater than the value in $/proc/sys/net/core/somaxconn$, then it is silently capped to that value.[85]

$connect(2)$

The $connect(2)$ system call starts the link between a client and a specific IP address and port number. The connection would complete when the other side will call the $accept(2)$ system call.

<u>From $man(2)$</u>:

The $int\ connect(int\ sockfd, const\ struct\ sockaddr\ *addr, socklen\_t\ addrlen)$ system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. The $addrlen$ argument specifies the size of $addr$. The format of the address in $addr$ is determined by the address space of the socket $sockfd$.[86] If the socket $sockfd$ is of type $SOCK\_DGRAM$, then $addr$ is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type $SOCK\_STREAM$ or $SOCK\_SEQPACKET$, this call attempts to make a connection to the socket that is bound to the address specified by $addr$.

<u>Protocol specifics</u>:

- Some protocol sockets (e.g., UNIX domain stream sockets) may successfully $connect()$ only once.
- Some protocol sockets (e.g., datagram sockets in the UNIX and Internet domains) may use $connect()$ multiple times to change their association.
- Some protocol sockets (e.g., TCP sockets as well as datagram sockets in the UNIX and Internet domains) may dissolve the association by connecting to an address with the sa_family member of $sockaddr$ set to AF_UNSPEC; thereafter, the socket can be connected to another address.[87]

[84] See $tcp(7)$ for more information.
[85] Since Linux 5.4, the default in this file is 4096; in earlier kernels, the default value is 128. In kernels before 2.4.25, this limit was a hard coded value, $SOMAXCONN$, with the value 128.
[86] See $socket(2)$ for further details.
[87] AF_UNSPEC is supported on Linux since kernel 2.2.

$accept(2)$

The $accept(2)$ system call accepts the incoming stream connection request from the other side, by the listening socket. This system call completes the three-way handshake of stream socket (like TCP) and opens a new socket that has a direct communication channel only with the other party in the handshake.

From $man(2)$:

The $int\ accept(int\ sockfd, struct\ sockaddr\ *restrict\ addr, socklen\_t\ *restrict\ addrlen)$ system call is used with connection-based socket types.[88] It extracts the first connection request on the queue of pending connections for the listening socket, $sockfd$, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket $sockfd$ is unaffected by this call.

Parameters:

- The argument $sockfd$ is a socket that has been created with $socket(2)$, bound to a local address with $bind(2)$, and is listening for connections after a $listen(2)$.
- The argument $addr$ is a pointer to a $sockaddr$ structure. This structure is filled in with the address of the peer socket, as known to the communications layer. The exact format of the address returned $addr$ is determined by the socket's address family.[89] When $addr$ is NULL, nothing is filled in; in this case, $addrlen$ is not used and should also be NULL.
- The $addrlen$ argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by $addr$; on return it will contain the actual size of the peer address.

The returned address is truncated if the buffer provided is too small; in this case, $addrlen$ will return a value greater than was supplied to the call. If no pending connections are present on the queue, and the socket is not marked as nonblocking, $accept()$ blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections are present on the queue, $accept()$ fails with the error $EAGAIN$ or $EWOULDBLOCK$. In order to be notified of incoming connections on a socket, you can use $select(2)$, $poll(2)$, or $epoll(7)$. A readable event will be delivered when a new connection is attempted, and you may then call $accept()$ to get a socket for that connection. Alternatively, you can set the socket to deliver $SIGIO$ when activity occurs on a socket.[90]

[88] $SOCK\_STREAM$ or $SOCK\_SEQPACKET$.
[89] See $socket(2)$ and the respective protocol man pages.
[90] See $socket(7)$ for details.

$send(2)$ and $sendto(2)$

The $send(2)$ and $sendto(2)$ system calls are an analogy for the $write(2)$ system call, specifcly for sockets, with flags set to 0.[91] Those system calls return the number of bytes that were send trough the socket itself, as the number of bytes can vary. The $send(2)$system call designed for stream sockets (like TCP), while the $sendto(2)$ system call designed for datagram connection-less sockets (like UDP). When the $send(2)$ system call returns 0, it means that the other party closed the connection.

From $man(2)$:

The system calls $send(int\ sockfd, const\ void\ *buf, size\_t\ len, int\ flags)$ and $sendto(int\ sockfd, const\ void\ *buf, size\_t\ len, int\ flags, const\ struct\ sockaddr\ *dest\_addr, socklen\_t\ addrlen)$ are used to transmit a message to another socket. The $send()$ call may be used only when the socket is in a connected state.[92] The only difference between $send()$ and $write(2)$ is the presence of flags. With a zero flags argument, $send()$ is equivalent to $write(2)$.

Also, the following call:

$$send(sockfd, buf, len, flags);$$

is equivalent to:

$$sendto(sockfd, buf, len, flags, NULL, 0);$$

The argument $sockfd$ is the file descriptor of the sending socket. If $sendto()$ is used on a connection-mode[93] socket, the arguments $dest\_addr$ and $addrlen$ are ignored,[94] and the error ENOTCONN is returned when the socket was not actually connected. Otherwise, the address of the target is given by $dest\_addr$ with $addrlen$ specifying its size. For $send()$ and $sendto()$, the message is found in buf and has length len.

If the message is too long to pass atomically through the underlying protocol, the error $EMSGSIZE$ is returned, and the message is not transmitted.[95] No indication of failure to deliver is implicit in a $send()$. Locally detected errors are indicated by a return value of -1. When the message does not fit into the send buffer of the socket, $end()$ normally blocks, unless the socket has been placed in nonblocking I/O mode. In nonblocking mode, it would fail with the error $EAGAIN$ or $EWOULDBLOCK$ in this case. The $select(2)$ call may be used to determine when it is possible to send more data.

[91] In this course we won't touch the subject of flags.
[92] So that the intended recipient is known.
[93] $SOCK\_STREAM$ or $SOCK\_SEQPACKET$.
[94] And the error $EISCONN$ may be returned when they are not NULL and 0.
[95] In TCP, the max transmission unit is 64KB. In UDP, the max transmission unit is 65,507 bytes theoretical, and the MTU is 1500 bytes.

$recv(2)$ and $recvfrom(2)$

The $recv(2)$ and $recvfrom(2)$ system calls are an analogy for the $read(2)$ system call, specifically for sockets, with flags set to 0.[96] Those system calls return the number of bytes that were received through the socket itself, as the number of bytes can vary. The $recv(2)$system call designed for stream sockets (like TCP), while the $recvfrom(2)$ system call designed for datagram connection-less sockets (like UDP). When the $recv(2)$ system call returns 0, it means that the other party closed the connection.

From $man(2)$:

The $recv(int\ sockfd, void\ *buf, size\_t\ len, int\ flags)$ and $recvfrom(int\ sockfd, void\ *restrict\ buf, size\_t\ len, int\ flags, struct\ sockaddr\ *restrict\ src\_addr, socklen\_t\ *restrict\ addrlen)$ calls are used to receive messages from a socket. They may be used to receive data on both connectionless and connection-oriented sockets. This page first describes common features of all three system calls, and then describes the differences between the calls. The only difference between $recv()$ and $read(2)$ is the presence of flags. With a zero flags argument, $recv()$ is generally equivalent to $read(2)$. Also, the following call:

$$recv(sockfd, buf, len, flags);$$

is equivalent to:

$$recvfrom(sockfd, buf, len, flags, NULL, NULL);$$

All three calls return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from. If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking,[97] in which case the value -1 is returned and errno is set to $EAGAIN$ or $EWOULDBLOCK$. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested. An application can use $select(2)$, $poll(2)$, or $epoll(7)$ to determine when more data arrives on a socket.

$recvfrom()$ places the received message into the buffer buf. The caller must specify the size of the buffer in len. If src_addr is not NULL, and the underlying protocol provides the source address of the message, that source address is placed in the buffer pointed to by src_addr. In this case, addrlen is a value-result argument. Before the call, it should be initialized to the size of the buffer associated with src_addr. Upon return, $addrlen$ is updated to contain the actual size of the source address. The returned address is truncated if the buffer provided is too small; in this case, $addrlen$ will return a value greater than was supplied to the call. If the caller is not interested in the source address, src_addr and $addrlen$ should be specified as NULL.

The $recv()$ call is normally used only on a connected socket.[98] It is equivalent to the call:

$$recvfrom(fd, buf, len, flags, NULL, 0);$$

[96] In this course we won't touch the subject of flags.
[97] See $fcntl(2)$.
[98] See $connect(2)$.

$close(2)$ and $shutdown(2)$

The $close(2)$ and $shutdown(2)$ system calls close the socket. While $close(2)$ system call closes the whole socket, the $shutdown(2)$ system call partially closes the socket for read/write.

From $man(2)$:

$close(int\ fd)$ closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks[99] held on the file it was associated with, and owned by the process, are removed.[100] If fd is the last file descriptor referring to the underlying open file description,[101] the resources associated with the open file description are freed; if the file descriptor was the last reference to a file which has been removed using $unlink(2)$, the file is deleted.

The $shutdown(int\ sockfd, int\ how)$ call causes all or part of a full-duplex connection on the socket associated with $sockfd$ to be shut down. The how paraments has three options:

- If how is $SHUT\_RD$, further receptions will be disallowed.
- If how is $SHUT\_WR$, further transmissions will be disallowed.
- If how is $SHUT\_RDWR$, further receptions and transmissions will be disallowed.[102]

Little Endian and Big Endian

Little and big endian are two ways of storing multibyte datatypes (int, float, etc.). In little endian machines, last byte of binary representation of the multibyte datatype is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte datatype is stored first.

Most of the times compiler takes care of endianness, however, endianness can become an issue. It matters in network programming: Suppose you write integers to file on a little-endian machine, and you transfer this file to a big-endian machine. Unless there is little endian to big endian transformation, big endian machine will read the file in reverse order. Standard byte order for networks is big endian, also known as network byte order. Before transferring data on network, data is first converted to network byte order (big endian). Sometimes it matters when you are using type casting.[103]

Sockets on Windows

While working with sockets in Linux is straightforward, in Windows you first need to include this piece of code, that forces the use of WinSock API:

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>    // Windows C standard library
#include <winsock2.h>   // WinSock library
int main(void) {
        WSADATA wsaData;
        if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
                fprintf(stderr, "WSAStartup failed.\n");
                exit(1);
        }
        else if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2) {
                fprintf(stderr, "Version 2.2 of WinSock isn't available.\n");
                WSACleanup();
                exit(2);
        }
        // Rest of the code...
}
```

[99] See $fcntl(2)$.
[100] Regardless of the file descriptor that was used to obtain the lock.
[101] See $open(2)$.
[102] Same as $close(2)$.
[103] You learned most of this subject in Computer Architecture and Computer Networks courses.

Lecture 5

Blocking I/O operations

There are functions and system calls that return. Some of those functions and system calls return only when external event happened ("good event"). Those functions are called "blocking I/O functions" or "blocking" for short. Examples:

- The $pow(3)$ library function returns when it finishes the mathematical calculation.
- The $scanf(3)$ library function will only return when the user inputs something from the standard input (stdin). If the user doesn't input anything, the function will never return and will "stuck" the program itself.
- The system call $wait(2)$ of the parent process, it can get stuck as in theory **the child process could never stop running**.

Question: What's the problem with blocking I/O functions?
Answer: The main problem is that we can't work parallel. For example, if you have a server which provides some service to clients using the $recv(2)$ system call to listen to request, the server will get stuck and won't be able to do other things.

Possible solutions – There are some possible solutions, none of them solves the problem completely:

- **Possible solution #1:** Opening few child processes using the $fork(2)$ system call, so each child process will handle a different client.
  The problem: We create a lot of processes and its resource expensive. Also, the system interacts with the clients simultaneously, which requires a shared memory, which doesn't work with different processes without extra sockets and overhead.
- **Possible solution #2:** Using threads, that use a shared memory.
  The problem: We didn't learn it; we learn about threads in the following lectures. Also, it is still expensive.
- **Possible solution #3:** Changning the flags of the file descriptor itself to be a non-blocking file descriptor, meaning if we call to $read(2)$ or $receive(2)$ system calls, they will return immediately if no data was received.
  The problem: It only partly solves the problem and creates new ones: If we have a few sockets that all are non-blocking we can't know for sure if one of the sockets is returned because it has data to read or because it has no data. If we check them again and again in a big while loop to all the clients, it will make the CPU usage to the maximal usage without actually doing anything useful,[104] which chocks other processes in the system.

To solve this problem, we will use I/O Multiplexing, which include system calls that help us get only the "hot" file descriptors, meaning only the one that are guaranteed to return immediately because they have data to be read, and thus we can handle them accordingly. Meaning, we want to run a function that will observe a set of file descriptors and block until one of the file descriptors is ready to be read. And thus, in this lecture, we will talk about the $select(2)$ and $poll(2)$ system calls that are designed to overcome this problem.

[104] As it's the same as driving in neutral and pushing the throttle to maximum power, asking "Did you send me anything?" over and over.

$select(2)$[105]

The $select(2)$ system call is an old API of POSIX, comparing to the newer $poll(2)$ system call API of Linux. The $select(2)$ API receives 3 sets of file descriptors and return the "hot" file descriptors.

The $select(2)$ system call receive 5 parameters. The first parameter tells us how many file descriptors we want to handle. The last parameter is the timeout value of which the $select(2)$ system call will block and return if there were not any "hot" file descriptors. The function returns an integer that represents the number of "hot" file descriptors, 0 if there was a timeout or -1 in case of an error.[106]

The $select(2)$ API requires including the following header files:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
```

The system call function structure is:

$$int\ select(int\ numfds, fd\_set\ *readfds, fd\_set\ *writefds, fd\_set\ *exceptfds, struct\ timeval\ *timeout);$$

<u>Arguments breakdown</u>:

- $numfds$ – The highest file descriptor id + 1.
- $readfds$ – A set of file descriptors that are intended to be read from.
- $writefs$ – A set of file descriptors that are intended to be written to.
- $execeptfds$ – A set of file descriptors that are expected to throw errors.
- timeout – A $timeval$ struct that defines the time that the $select(2)$ API will wait until it will return if there are no "hot" file descriptors.
  The $timeval$ struct is defined as the following:

```
struct timeval {
        int tv_sec;
        int tv_usec;
};
```

  <u>Struct breakdown</u>:
  - tv_sec – The number of seconds we want to wait.
  - tv_usec – The number of microseconds we want to wait.

For the $select(2)$, there are some important macros that are used to manage and handle the file descriptors:

Macro	Description
$FD\_SET(int\ fd, fd\_set\ *\ set)$	Put a specific file descriptor into a specific set.
$FD\_CLR(int\ fd, fd\_set\ *\ set)$	Removes a specific file descriptor from a specific set.
$FD\_ISSE(int\ fd, fd\_set\ *\ set)$	Returns true if a given file descriptor is in a given set, false otherwise.
$FD\_ZERO(fd\_set\ *\ set)$	Clears a given set of file descriptors and set it to the empty set.

[105] **<u>Warning</u>:** $select(2)$ can monitor only file descriptors numbers that are less than $FD\_SETSIZE$ (1024)—an unreasonably low limit for many modern applications—and this limitation will not change. All modern applications should instead use $poll(2)$ or $epoll(7)$, which do not suffer this limitation.
[106] The $select(2)$ system call is problematic because some of the parameters are intendent for input and output, meaning it overrides the parameters that it gets, so we need to save them.

Basic example for usage: Waiting 2.5 seconds for the user to input something.[107]

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN                   0                       // File descriptor for standard input

int main(void) {
        struct timeval tv;
        fd_set readfds;

        // Configure timeout to be 2.5 seconds.
        tv.tv_sec = 2;
        tv.tv_usec = 500000;

        FD_ZERO(&readfds);                              // Initialize and reset the set
        FD_SET(STDIN, &readfds);                        // Adding the standard input (stdin) to the set

        // Don't care about writefds and exceptfds:
        select(STDIN + 1, &readfds, NULL, NULL, &tv);

        if (FD_ISSET(STDIN, &readfds)) {                // Check if the stdin file descriptor is "hot".
                fprintf(stdout, "A key was pressed!\n");
        }

        else {
                fprintf(stdout, "Timed out.\n");
        }

        return 0;
}
```

For more complex example of a server using the $select(2)$ API, **click here**.[108]

From $man(2)$:

$select(int\ nfds, fd\_set\ *restrict\ readfds, fd\_set\ *restrict\ writefds, fd\_set\ *restrict\ exceptfds, struct\ timeval\ *restrict\ timeout)$ allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation.[109] A file descriptor is considered ready if it is possible to perform a corresponding I/O operation[110] without blocking.

File descriptor sets:

The principal arguments of $select()$ are three "sets" of file descriptors,[111] which allow the caller to wait for three classes of events on the specified set of file descriptors. Each of the fd_set arguments may be specified as NULL if no file descriptors are to be watched for the corresponding class of events.[112]

---

[107] Something will appear in the standard input.
[108] Beej's Guide to Network Programming example called $selectserver.c$.
[109] e.g., input possible.
[110] e.g., $read(2)$, or a sufficiently small $write(2)$.
[111] Declared with the type fd_set.
[112] Upon return, each of the file descriptor sets is modified in place to indicate which file descriptors are currently "ready". Thus, if using $select()$ within a loop, the sets must be reinitialized before each call.

The contents of a file descriptor set can be manipulated using the following macros:

- $FD\_ZERO(fd\_set\ *set)$ – This macro clears[113] set. It should be employed as the first step in initializing a file descriptor set.
- $FD\_SET(int\ fd, fd\_set\ *set)$ – This macro adds the file descriptor fd to set. Adding a file descriptor that is already present in the set is a no-op and does not produce an error.
- $FD\_CLR(int\ fd, fd\_set\ *set)$ – This macro removes the file descriptor fd from set. Removing a file descriptor that is not present in the set is a no-op and does not produce an error.
- $FD\_ISSET(int\ fd, fd\_set\ *set)$ – $select()$ modifies the contents of the sets according to the rules described below. After calling $select()$, the $FD\_ISSET()$ macro can be used to test if a file descriptor is still present in a set. $FD\_ISSET()$ returns nonzero if the file descriptor fd is present in set, and zero if it is not.

<u>The arguments of $select()$ are as follows</u>:

- $readfds$ – The file descriptors in this set are watched to see if they are ready for reading. A file descriptor is ready for reading if a read operation will not block; in particular, a file descriptor is also ready on end-of-file. After $select()$ has returned, $readfds$ will be cleared of all file descriptors except for those that are ready for reading.
- $writefds$ – The file descriptors in this set are watched to see if they are ready for writing. A file descriptor is ready for writing if a write operation will not block. However, even if a file descriptor indicates as writable, a large write may still block. After $select()$ has returned, $writefds$ will be cleared of all file descriptors except for those that are ready for writing.
- $exceptfds$ – The file descriptors in this set are watched for "exceptional conditions".[114] After $select()$ has returned, $exceptfds$ will be cleared of all file descriptors except for those for which an exceptional condition has occurred.
- $nfds$ – This argument should be set to the highest-numbered file descriptor in any of the three sets, plus 1. The indicated file descriptors in each set are checked, up to this limit.
- $timeout$ – The timeout argument is a $timeval$ structure that specifies the interval that $select()$ should block waiting for a file descriptor to become ready. The call will block until either:
    - A file descriptor becomes ready.
    - The call is interrupted by a signal handler.
    - The timeout expires.

Note that the timeout interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount. If both fields of the $timeval$ structure are zero, then $select()$ returns immediately.[115] If timeout is specified as NULL, $select()$ blocks indefinitely waiting for a file descriptor to become ready.

[113] Removes all file descriptors from.
[114] For examples of some exceptional conditions, see the discussion of $POLLPRI$ in $poll(2)$.
[115] This is useful for polling.

$poll(2)$

The $poll(2)$ system call API is a newer version of the $select(2)$ system call API, which is more comfortable to use and doesn't have the limitations of the $select(2)$ API. The main difference is that $poll(2)$ doesn't receive a set and doesn't override the arguments.

To use the $poll(2)$ API we need to including the following header file:

```
#include <poll.h>
```

The structure of the $poll(2)$ system call is:

$$int\ poll(struct\ pollfd\ fds[], nfds\_t\ nfds, int\ timeout);$$

Arguments breakdown:

- fds – An array of $pollfd$ structs that contains the relevant file descriptors.
  The poll struct is defined as the following:

```
struct pollfd {
        int fd;
        short events;
        short revents;
};
```

  Struct breakdown:
  - fd - The file descriptor itself.
  - events - The relevant events that the $poll(2)$ API will check for that specific file descriptor. It's an OR bitwise between $POLLIN$ for reading and $POLLOUT$ for writing.
  - $revents$ - An OR bitwise of all the events that were occurred when the $poll(2)$ system call returned, for that file descriptor.
- $nfds$ - The size of the fds array, or the number of the file descriptor we care about.
- timeout - The time in milliseconds that the $poll(2)$ API will wait until it will return if there are no "hot" file descriptors.

Basic example for usage: Waiting 2.5 seconds for the user to input something.[116]

```
#include <stdio.h>
#include <poll.h>
int main(void) {
        struct pollfd pfds[1];                  // More if you want to monitor more
        pfds[0].fd = 0;                         // Standard input
        pfds[0].events = POLLIN;                // Tell me when ready to read
        // If you needed to monitor other things, as well:
        //pfds[1].fd = some_socket;             // Some socket descriptor
        //pfds[1].events = POLLIN;              // Tell me when ready to read
        printf("Hit RETURN or wait 2.5 seconds for timeout\n");
        int num_events = poll(pfds, 1, 2500);   // 2.5 second timeout
        if (num_events == 0) {
                printf("Poll timed out!\n");
        } else {
                int pollin_happened = pfds[0].revents & POLLIN;
                if (pollin_happened) {
                        printf("File descriptor %d is ready to read\n", pfds[0].fd);
                } else {
                        printf("Unexpected event occurred: %d\n", pfds[0].revents);
                }
        }
        return 0;
}
```

[116] Something will appear in the standard input.

From $man(2)$:

$poll(struct\ pollfd\ *fds, nfds\_t\ nfds, int\ timeout)$ performs a similar task to $select(2)$: it waits for one of a set of file descriptors to become ready to perform I/O.[117] The set of file descriptors to be monitored is specified in the fds argument, which is an array of $pollfd$ structures. The caller should specify the number of items in the fds array in $nfds$.

$pollfd$ struct fields:

- The field fd contains a file descriptor for an open file. If this field is negative, then the corresponding events field is ignored and the $revents$ field returns zero.[118]
- The field $events$ is an input parameter, a bit mask specifying the events the application is interested in for the file descriptor fd. This field may be specified as zero, in which case the only events that can be returned in $revents$ are $POLLHUP$, $POLLERR$, and $POLLNVAL$.
- The field $revents$ is an output parameter, filled by the kernel with the events that actually occurred. The bits returned in $revents$ can include any of those specified in events, or one of the values $POLLERR$, $POLLHUP$, or $POLLNVAL$.[119]

If none of the events requested (and no error) has occurred for any of the file descriptors, then $poll()$ blocks until one of the events occurs. The timeout[120] argument specifies the number of milliseconds that $poll()$ should block waiting for a file descriptor to become ready. The call will block until either:

- A file descriptor becomes ready.
- The call is interrupted by a signal handler.
- The timeout expires.

The bits that may be set/returned in events and $revents$ are defined in $< poll.h >$ header file:

- **POLLIN** – There is data to read.
- **POLLPRI** – There is some exceptional condition on the file descriptor. Possibilities include:
  - There is out-of-band data on a TCP socket.[121]
  - A pseudo terminal master in packet mode has seen a state change on the slave.[122]
  - A $cgroup.events$ file has been modified.[123]

---

[117] The Linux-specific $epoll(7)$ API performs a similar task but offers features beyond those found in $poll()$.
[118] This provides an easy way of ignoring a file descriptor for a single $poll()$ call: simply negate the fd field. Note, however, that this technique can't be used to ignore file descriptor 0.
[119] These three bits are meaningless in the events field and will be set in the $revents$ field whenever the corresponding condition is true.
[120] Note that the timeout interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount. Specifying a negative value in timeout means an infinite timeout. Specifying a timeout of zero causes $poll()$ to return immediately, even if no file descriptors are ready.
[121] See $tcp(7)$.
[122] See $ioctl\_tty(2)$.
[123] See $cgroups(7)$.

- **POLLOUT** – Writing is now possible, though a write larger than the available space in a socket or pipe will still block.[124]
- **POLLRDHUP**[125] – Stream socket peer closed connection or shut down writing half of connection.[126]
- **POLLERR** – Error condition.[127] This bit is also set for a file descriptor referring to the write end of a pipe when the read end has been closed.
- **POLLHUP** – Hang up.[128] Note that when reading from a channel such as a pipe or a stream socket, this event merely indicates that the peer closed its end of the channel. Subsequent reads from the channel will return 0 (end of file) only after all outstanding data in the channel has been consumed.
- **POLLNVAL** – Invalid request: fd not open.[129]

[124] Unless $O\_NONBLOCK$ is set.
[125] Since Linux 2.6.17.
[126] The $\_GNU\_SOURCE$ feature test macro must be defined (before including any header files) in order to obtain this definition.
[127] Only returned in $revents$; ignored in events.
[128] Only returned in $revents$; ignored in events.
[129] Only returned in $revents$; ignored in events.

Lecture 6

Introduction to threads

Whenever there is process,[130] all the work is done on that process alone, and it doesn't refer to other processes. We could say that the process is independent from others, does only his own tasks and has access only to his own memory.

When we need to handle parallel and continuous processes, there are few solutions to overcome the problem. One solution is to use a shared memory for all the processes, meaning all the processes have access to the same memory space address. It this case, processes could read infromation from others and once there is a specific connection, all the processes can see the transferred data.

In a situation where there is a thread that was created, which manages the sub-process,[131] any process can read its section in the shared memory at its own pace.

We can conclude that there are processes that act independently, and there are processes that are connected to other processes and work in a shared context. The connected processes read the shared memory at their own pace and each one does its task, while not violating the shared memory and not changing the behavior of their own actions.

If there are a bunch of threads that work in parallel and need to solve a shared problem, we can use one shared memory space that contains a lot of pointers. Each pointer is used in a specific thread and is located on different memory address. Each pointer codecs certain actions that are needed to be done in runtime.

Using the many pointers that are located at the shared memory space, we can harvest in ordered way the tasks that each thread must do. The pointers know how to run the tasks in parallel, while keeping the shared data safe and avoiding accidental violate access or corruption of the shared data.

In general, the advantage for this method is that we can use parallel calculation to solve shared problems, while the usage of private memory spaces could create problems like accessing to the shared data and resistance between two threads.

In the inner instructions of the program, each thread is allocated with its own stack memory, which is his own private memory space for executing his code. When a new thread is created, we need to allocate memory for its stack, by using the $malloc(3)$ library function.

Nevertheless, the main problem with the threads is that each thread does have access to other threads stack memory, and have the privileges to change the data there, potentially corrupting it, and thus it isn't a smart way to program.

To understand this technique more clearly, we can summarize that:

- Each thread gets its own private stack memory.
- Threads are allocated stack memory using the $malloc(3)$ library function.
- Each thread can access other threads stack memory and changing the values there.
- This action can create an undefine behavior and thus isn't recommended.

[130] If we didn't define something else.
[131] With the counter pointer.

So, to summarize, threads have shared memory space, and each thread has his own private stack memory, but other threads can still access it to allow other threads to change it.

It's very common nowadays to work with threads because it's easier to encode compering to a process. Does it mean that processes are obsolete? No, and here are some reasons why we still need processes:

- **When writing a debugger** – There is a process that we want to debug, and is a "suspicious" process, we want that our program will continue to run even if the process died, because we want to inspect it from the outside. If we want to run the program externally without causing the debugger to exit, we need to run the debugger in a completely different process.
- **Customer service website** – Today there aren't databases servers that are also web servers and vice versa. Why can't we join them together? For cyber security purposes, web servers are more prone to cyber-attacks, and thus we'll separate those layers.
- When two companies want to run a separate process on two separate things, we don't want the data to be mixed.

<u>The efficiency of threads</u>:

Suppose that there is a document that while writing to it, we want to show spelling mistakes and in parallel to back up the document to the hard drive. All this can be done in one process, but the decision making will become very complex, and the transitions will become expensive on resources. We can also create a bunch of processes, but context switching is still expensive on resources. If there is only one interactive task it's better let only one process to handle it, but if there is a task that requires interaction, we'll benefit if we'll work with threads.

<u>Advantages of threads</u>:

Threads share a memory space, can communicate between themselves efficiently and it's easy to create and destroy them. Since all the threads share the same memory, if a memory of one thread gets corrupted, all the threads get corrupted and die.[132] They share data, code, open file descriptors, signal table, but each one also has a private stack memory space. When creating a thread, we can set its attributes.[133]

In this course we will use the pthread library that gives use support for threads. When we compile, we must add the following flags:

$$-pthread$$

If we won't compile with this flag, the behavior of the pthread library function is undefined.

[132] Same as diving by zero.
[133] For example: the size of the stack, how much CPU time it could use, etc.

$pthread\_create(3)$

This library function creates and runs a new thread with given attributes and a start routine function.

From $man(3)$:

The $pthread\_create(pthread\_t\ *restrict\ thread, const\ pthread\_attr\_t\ *restrict\ attr, void\ *(*start\_routine)(void\ *), void\ *restrict\ arg)$ function starts a new thread in the calling process. The new thread starts execution by invoking $start\_routine()$; arg is passed as the sole argument of $start\_routine()$.

The new thread terminates in one of the following ways:

- It calls $pthread\_exit(3)$, specifying an exit status value that is available to another thread in the same process that calls $pthread\_join(3)$.
- It returns from $start\_routine()$. This is equivalent to calling $pthread\_exit(3)$ with the value supplied in the return statement.
- It is canceled.[134]
- Any of the threads in the process calls $exit(3)$, or the main thread performs a return from $main()$.[135]

The $attr$ argument points to a $pthread\_attr\_t$ structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using $pthread\_attr\_init(3)$ and related functions. If $attr$ is NULL, then the thread is created with default attributes. Before returning, a successful call to $pthread\_create()$ stores the ID of the new thread in the buffer pointed to by thread; this identifier is used to refer to the thread in subsequent calls to other pthreads functions. The new thread inherits a copy of the creating thread's signal mask.[136] The set of pending signals for the new thread is empty.[137] The new thread does not inherit the creating thread's alternate signal stack.[138] The new thread inherits the calling thread's floating-point environment.[139] The initial value of the new thread's CPU-time clock is 0.[140]

$pthread\_cancel(3)$

This library function tries to cancel a thread but doesn't guarantee that the thread will stop.

From $man(3)$:

The $pthread\_cancel(pthread\_t\ thread)$ function sends a cancellation request to the thread $thread$. Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: its cancelability state and type.

---

[134] See $pthread\_cancel(3)$.
[135] This causes the termination of all threads in the process.
[136] See $pthread\_sigmask(3)$.
[137] See $sigpending(2)$.
[138] See $sigaltstack(2)$.
[139] See $fenv(3)$.
[140] See $pthread\_getcpuclockid(3)$.

A thread's cancelability state, determined by $pthread\_setcancelstate(3)$, can be enabled[141] or disabled. If a thread has disabled cancellation, then a cancellation request remains queued until the thread enables cancellation. If a thread has enabled cancellation, then its cancelability type determines when cancellation occurs.

A thread's cancellation type, determined by $pthread\_setcanceltype(3)$, may be either asynchronous or deferred.[142] Asynchronous cancelability means that the thread can be canceled at any time.[143] Deferred cancelability means that cancellation will be delayed until the thread next calls a function that is a cancellation point.[144]

When a cancellation requested is acted on, the following steps occur for thread:[145]

1. Cancellation clean-up handlers are popped[146] and called.[147]
2. Thread-specific data destructors are called, in an unspecified order.[148]
3. The thread is terminated.[149]

The above steps happen asynchronously with respect to the $pthread\_cancel()$ call; the return status of $pthread\_cancel()$ merely informs the caller whether the cancellation request was successfully queued. After a canceled thread has terminated, a join with that thread using $pthread\_join(3)$ obtains $PTHREAD\_CANCELED$ as the thread's exit status.[150]

$pthread\_join(3)$

This library function joins the given thread with the thread that is called the join function. The caller thread will wait until the joined thread has finished running.

From $man(3)$:

The $pthread\_join(pthread\_t\ thread, void\ **\ retval)$ function waits for the thread specified by thread to terminate. If that thread has already terminated, then $pthread\_join()$ returns immediately. The thread specified by thread must be joinable.

If $retval$ is not NULL, then $pthread\_join()$ copies the exit status of the target thread[151] into the location pointed to by retval. If the target thread was canceled, then $PTHREAD\_CANCELED$ is placed in the location pointed to by retval.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling $pthread\_join()$ is canceled, then the target thread will remain joinable.[152]

---

[141] The default for new threads.
[142] The default for new threads.
[143] Usually immediately, but the system does not guarantee this.
[144] A list of functions that are or may be cancellation points is provided in $pthreads(7)$.
[145] In this order.
[146] In the reverse of the order in which they were pushed.
[147] See $pthread\_cleanup\_push(3)$.
[148] See $pthread\_key\_create(3)$.
[149] See $pthread\_exit(3)$.
[150] Joining with a thread is the only way to know that cancellation has completed.
[151] i.e., the value that the target thread supplied to $pthread\_exit(3)$.
[152] i.e., it will not be detached.

$pthread\_detach(3)$

This library function detaches a thread from the processes, meaning that it no longer belongs to the original process, and thus can't be joined anymore.

From $man(3)$:

The $pthread\_detach(pthread\_t\ thread)$ function marks the thread identified by thread as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread. Attempting to detach an already detached thread results in unspecified behavior.

$pthread\_exit(3)$

This library function forces the called thread to exit.

From $man(3)$:

The $pthread\_exit(void\ *retval)$ function terminates the calling thread and returns a value via $retval$ that[153] is available to another thread in the same process that calls $pthread\_join(3)$.

Any clean-up handlers established by $pthread\_cleanup\_push(3)$ that have not yet been popped, are popped[154] and executed. If the thread has any thread-specific data, then, after the clean-up handlers have been executed, the corresponding destructor functions are called, in an unspecified order. When a thread terminates, process-shared resources[155] are not released, and functions registered using $atexit(3)$ are not called. After the last thread in a process terminates, the process terminates as by calling $exit(3)$ with an exit status of zero; thus, process-shared resources are released, and functions registered using $atexit(3)$ are called.

$pthread\_kill(3)$

This library function sends a signal to specific thread.

From $man(3)$:

The $pthread\_kill(pthread\_t\ thread, int\ sig)$ function sends the signal sig to thread, a thread in the same process as the caller. The signal is asynchronously directed to thread. If sig is 0, then no signal is sent, but error checking is still performed.

$pthread\_equal(3)$

This library function returns if two threads are the same.

From $man(3)$:

The $pthread\_equal()$ function compares two thread identifiers.

---

[153] If the thread is joinable.
[154] In the reverse of the order in which they were pushed.
[155] e.g., mutexes, condition variables, semaphores, and file descriptors.

$pthread\_self(3)$

This library function returns the thread identification of the calling thread.

From $man(3)$:

The $pthread\_self()$ function returns the ID of the calling thread. This is the same value that is returned in $*thread$ in the $pthread\_create(3)$ call that created this thread.

$pthread\_attr\_init(3)$ and $pthread\_attr\_destroy(3)$

Those library functions initialize and destroy the attributes variable of a thread. With those library functions we can change the thread attribute before creating it.

From $man(3)$:

The $pthread\_attr\_init(pthread\_attr\_t\ *attr)$ function initializes the thread attributes object pointed to by $attr$ with default attribute values. After this call, individual attributes of the object can be set using various related functions,[156] and then the object can be used in one or more $pthread\_create(3)$ calls that create threads. Calling $pthread\_attr\_init()$ on a thread attributes object that has already been initialized results in undefined behavior.

When a thread attributes object is no longer required, it should be destroyed using the $pthread\_attr\_destroy(pthread\_attr\_t\ *attr)$ function. Destroying a thread attributes object has no effect on threads that were created using that object. Once a thread attributes object has been destroyed, it can be reinitialized using $pthread\_attr\_init()$. Any other use of a destroyed thread attributes object has undefined results.

Attribute functions:

- $pthread\_attr\_setaffinity\_np(3)$ – Set CPU affinity attribute.
- $pthread\_attr\_setdetachstate(3)$ – Set the detach state attribute.
- $pthread\_attr\_setguardsize(3)$ – Set guard size attribute.
- $pthread\_attr\_setinheritsched(3)$ – Set inherit-scheduler attribute.
- $pthread\_attr\_setschedparam(3)$ – Set scheduling parameter attributes.
- $pthread\_attr\_setschedpolicy(3)$ – Set scheduling policy attribute.
- $pthread\_attr\_setscope(3)$ – Set contention scope attribute.
- $pthread\_attr\_setsigmask\_np(3)$ – Set signal mask attribute.
- $pthread\_attr\_setstack(3)$ – Set stack attributes.
- $pthread\_attr\_setstackaddr(3)$ – Set stack address attribute.
- $pthread\_attr\_setstacksize(3)$ – Set stack size attribute.
- $pthread\_getattr\_np(3)$ – Get attributes of created thread.
- $pthread\_setattr\_default\_np(3)$ – Set default thread-creation attributes.

[156] Listed under attribute functions.

$pthread\_atfork(3)$

This library function register fork handlers when we call the $fork(2)$ system call inside of a thread.

From $man(3)$:

The $pthread\_atfork(void\ (*\ prepare)(void), void\ (*\ parent)(void), void\ (*\ child)(void))$ function registers fork handlers that are to be executed when $fork(2)$ is called by this thread. The handlers are executed in the context of the thread that calls $fork(2)$.

Three kinds of handler can be registered:

- Prepare specifies a handler that is executed before $fork(2)$ processing starts.
- Parent specifies a handler that is executed in the parent process after $fork(2)$ processing completes.
- Child specifies a handler that is executed in the child process after $fork(2)$ processing completes.

Any of the three arguments may be NULL if no handler is needed in the corresponding phase of $fork(2)$ processing.

Thread-Specific Storage (TSS)

During the course, we learned that in processes the memory space is separated for each process, while in threads it is shared. There are some situations where we want to share memory also in the context of multiple threads. One such way that we learned for this purpose is using memory mapping ($mmap$).[157]

Moreover, there are situations where we work with threads but don't want to share data between them. This case is rarer, and we won't go deep into the matter in this course, but I will still give you an example, to show that this situation can happen in real life scenario.

The address of the memory map allows processes to map files or devices to their virtual memory. A shared memory allows a bunch of processes to access the same memory area, which enables sharing data and efficient communicate between them. Instead of coping data between processes, they can read and write directly to the shared memory space, and thus and thus avoid the need for data duplication.

Memory mapping is designed for situations where we need to share memory between processes, but as stated, there are situations where we want to work with threads without sharing the memory. One of those situations is when we have specific variables in the code that are only meant to be accessed to the thread that executes them and no other threads. An example of this kind of variable is the $errno$ variable.

When the code is executed and is doing system calls to the operating system, the operating system can return an error code using the $errno$ variable. The values of errno are defined in header files like $errno.h$ in C and C++, and each value represents a differnet error code. Programmers can use the $errno$ variable to get the correct error code to the function or the last system call that was executed by the operating system, and to handle it accordingly. If we

[157] There are more APIs that are out of scope for this course.

compare the value of $errno$ to the predefined error codes, we can know what's the type of the error is and handle it accordingly.

The memory mapping and the $errno$ variable are examples of usage of the Thread Specific Storage (TSS). Each thread in the system can relate that errno value to a system call that the thread executed, and each thread can't access to this value on other threads. However, in situations where we need a variable that shares a common name between all the threads, but each thread assigns a different value, the TSS solution isn't compatible.

Thread synchronization

We'll now discuss two methods to do universal thread synchronization, which work on any programming language we want.

The first method is called “sock on door”, a term from the erotic world. It describes a situation where a shared resource like a room or in case of this course, a file, can be used by only one entity at a time. When an entity uses the resource, it “hangs a sock” on the door, which symbols to others that the resource is currently being used and they need to wait that the resource will become avaliable before trying to access it.

A more detailed version of the “sock on door” method:

1. A number of entities, like threads or processes, want to access a shared resource.
2. Before accessing the resource, the entity checks if there is “a sock hanging on the door”.
3. If there is, it means that someone else is using the resource and the entity must wait.
4. When an entity is finished using the resource, it “removes the sock” and states that the resource has been released and is now available.

In addition to ”sock on door”, there are also two locking mechanisms: Advisory lock and Mandatory lock. Each one is done differently and is appropriate to different situations.

The second method is "Producer-Consumer", which is a common synchronization pattern in parallel programming. In this pattern there are two components: the producer that produces data or tasks and the consumer which processes the created data or tasks. They are synchronized using a shared stock or queue.

Below is a sketch of the process:

1. The producers produce data or tasks and enqueues them to a shared queue.
2. Consumers dequeue the data or the tasks from the shared queue and process them.
3. The shared queue is used as a communication form between the producers and the consumers.
4. Appropriate synchronization mechanisms, not locks, are used to implement the producer-consumer pattern in a way that minimizes busy waiting.

Sync primitives are tools that are designed to support coordination and control of access to shared resources, and to ensure standard synchronization between threads or processes. They are essential components when dealing with concurrency, and they are used to prevent race conditions or other synchronization problems.

In conclusion, to synchronize between threads we can use the producer-consumer synchronization using the POSIX cond, or the “sock on door” synchronization using the POSIX mutex. To synchronize between processes, we can use producer-consumer synchronization using sockets, or "sock on door” synchronization using $fcntl(2)$ system call.[158]

To summarize, producer-consumer using POSIX cond (or sockets) and ”sock on door” using POSIX mutex are two different applications to manage threads and processes in the operating system.

Producer-consumer is a mechanism used in parallel programming and thread managing in the operating system. Using producer-consumer we can run bunch of threads in parallel and manage the flow of data between them. Instead of using sockets for communication between the threads, we can use POSIX cond, which provides communication and information sharing capabilities by using cond.

“Sock on door” with POSIX mutex is one technique that uses the block and wait issue to manage processes in an operating system. This is a basic and common usage where the process holds a skull (door) and waits for an event or further styling before continuing with the next operation. The sock on the door counts the number of processes and is common in the UNIX environment.

$fcntl(2)$

The $fcntl(2)$ system call is used to manipulate various things with the opened file descriptors in the system. This system call allows to make changes in the status of an opened file or a socket. In the context of process managing, $fcntl(2)$ allows us to lock or unlock locks on files and sockets, and thus able us to control the access of processes and threads to resources.

<u>From $man(2)$</u>:

$fcntl(int\ fd, int\ cmd, .../*\ arg\ */)$ is a file descriptor manipulator that performs one of the operations described below on the open file descriptor fd. The operation is determined by cmd. $fcntl()$ can take an optional third argument. Whether or not this argument is required is determined by cmd. The required argument type is indicated in parentheses after each cmd name,[159] or void is specified if the argument is not required.[160]

<u>Advisory record locking</u>

Linux implements traditional (“process-associated”) UNIX record locks, as standardized by POSIX. For a Linux-specific alternative with better semantics, see the discussion of open file description locks below. F_SETLK, F_SETLKW, and F_GETLK are used to acquire, release,

[158] Mandatory locks.
[159] In most cases, the required type is int, and we identify the argument using the name arg.
[160] Certain of the operations below are supported only since a particular Linux kernel version. The preferred method of checking whether the host kernel supports a particular operation is to invoke $fcntl()$ with the desired cmd value and then test whether the call failed with $EINVAL$, indicating, indicating that the kernel does not recognize this value.

and test for the existence of record locks.[161] The third argument, lock, is a pointer to a structure that has at least the following fields (in unspecified order).

The flock struct:

```
struct flock {
        ...
        short l_type;    /* Type of lock: F_RDLCK, F_WRLCK, F_UNLCK */
        short l_whence;  /* How to interpret l_start: SEEK_SET, SEEK_CUR, SEEK_END */
        off_t l_start;   /* Starting offset for lock */
        off_t l_len;     /* Number of bytes to lock */
        pid_t l_pid;     /* PID of process blocking our lock (set by F_GETLK and F_OFD_GETLK) */
        ...
};
```

The l_whence, l_start, and l_len fields of this structure specify the range of bytes we wish to lock. Bytes past the end of the file may be locked, but not bytes before the start of the file.

- l_start – The starting offset for the lock and is interpreted relative to either: the start of the file (if l_whence is $SEEK\_SET$); the current file offset (if l_whence is $SEEK\_CUR$); or the end of the file (if l_whence is $SEEK\_END$). In the final two cases, l_start can be a negative number provided the offset does not lie before the start of the file.
- l_len – Specifies the number of bytes to be locked. If l_len is positive, then the range to be locked covers bytes l_start up to and including $l\_start + l\_len - 1$. Specifying 0 for l_len has the special meaning: lock all bytes starting at the location specified by l_whence and l_start through to the end of file, no matter how large the file grows.[162]
- l_type – Can be used to place a read (F_RDLCK) or a write (F_WRLCK) lock on a file. Any number of processes may hold a read lock (shared lock) on a file region, but only one process may hold a write lock (exclusive lock). An exclusive lock excludes all other locks, both shared and exclusive. A single process can hold only one type of lock on a file region; if a new lock is applied to an already-locked region, then the existing lock is converted to the new lock type.[163]

Lock operations:

- F_SETLK ($struct\ flock\ *$) – Acquire a lock (when l_type is F_RDLCK or F_WRLCK) or release a lock (when l_type is F_UNLCK) on the bytes specified by the l_whence, l_start, and l_len fields of lock. If a conflicting lock is held by another process, this call returns -1 and sets errno to $EACCES$ or $EAGAIN$.[164]

[161] Also known as byte-range, file-segment, or file-region locks.
[162] POSIX.1-2001 allows (but does not require) an implementation to support a negative l_len value; if l_len is negative, the interval described by lock covers bytes $l\_start + l\_len$ up to and including l_start-1. This is supported by Linux since kernel versions 2.4.21 and 2.5.49.
[163] Such conversions may involve splitting, shrinking, or coalescing with an existing lock if the byte range specified by the new lock does not precisely coincide with the range of the existing lock.
[164] The error returned in this case differs across implementations, so POSIX requires a portable application to check for both errors.

- F_SETLKW ($struct\ flock\ *$) – As for F_SETLK, but if a conflicting lock is held on the file, then wait for that lock to be released. If a signal is caught while waiting, then the call is interrupted and[165] returns immediately.[166]
- F_GETLK ($struct\ flock\ *$) – On input to this call, lock describes a lock we would like to place on the file. If the lock could be placed, $fcntl()$ does not actually place it but returns F_UNLCK in the l_type field of lock and leaves the other fields of the structure unchanged.

If one or more incompatible locks would prevent this lock being placed, then $fcntl()$ returns details about one of those locks in the l_type, l_whence, l_start, and l_len fields of lock. If the conflicting lock is a traditional (process-associated) record lock, then the l_pid field is set to the PID of the process holding that lock. If the conflicting lock is an open file description lock, then l_pid is set to -1. Note that the returned information may already be out of date by the time the caller inspects it.

In order to place a read lock, fd must be open for reading. In order to place a write lock, fd must be open for writing. To place both types of lock, open a file read-write.

When placing locks with F_SETLKW, the kernel detects deadlocks, whereby two or more processes have their lock requests mutually blocked by locks held by the other processes.[167] If each process then attempts to lock the byte already locked by the other process using F_SETLKW, then, without deadlock detection, both processes would remain blocked indefinitely. When the kernel detects such deadlocks, it causes one of the blocking lock requests to immediately fail with the error $EDEADLK$; an application that encounters such an error should release some of its locks to allow other applications to proceed before attempting regain the locks that it requires. Circular deadlocks involving more than two processes are also detected.[168]

As well as being removed by an explicit F_UNLCK, record locks are automatically released when the process terminates. Record locks are not inherited by a child created via $fork(2)$, but are preserved across an $execve(2)$. Because of the buffering performed by the $stdio(3)$ library, the use of record locking with routines in that package should be avoided.[169]

The record locks described above are associated with the process. This has some unfortunate consequences:

- If a process closes any file descriptor referring to a file, then all of the process's locks on that file are released, regardless of the file descriptor(s) on which the locks were obtained.[170]
- The threads in a process share locks. In other words, a multithreaded program can't use record locking to ensure that threads don't simultaneously access the same region of a file.

---

[165] After the signal handler has returned.
[166] With return value -1 and errno set to $EINTR$; see $signal(7)$.
[167] For example, suppose process A holds a write lock on byte 100 of a file, and process B holds a write lock on byte 200.
[168] Note, however, that there are limitations to the kernel's deadlock-detection algorithm.
[169] Use $read(2)$ and $write(2)$ instead.
[170] This is bad: it means that a process can lose its locks on a file such as $/etc/passwd$ or $/etc/mtab$ when for some reason a library function decides to open, read, and close the same file.

Mandatory locking[171]

By default, both traditional (process-associated) and open file description record locks are advisory. Advisory locks are not enforced and are useful only between cooperating processes.[172]

Both lock types can also be mandatory. Mandatory locks are enforced for all processes. If a process tries to perform an incompatible access (e.g., $read(2)$ or $write(2)$) on a file region that has an incompatible mandatory lock, then the result depends upon whether the $O\_NONBLOCK$ flag is enabled for its open file description. If the $O\_NONBLOCK$ flag is not enabled, then the system call is blocked until the lock is removed or converted to a mode that is compatible with the access. If the $O\_NONBLOCK$ flag is enabled, then the system call fails with the error $EAGAIN$.

To make use of mandatory locks, mandatory locking must be enabled both on the filesystem that contains the file to be locked, and on the file itself. Mandatory locking is enabled on a filesystem using the "$-o\ mand$" option to $mount(8)$, or the $MS\_MANDLOCK$ flag for $mount(2)$. Mandatory locking is enabled on a file by disabling group execute permission on the file and enabling the set-group-ID permission bit.[173]

Lost locks

When an advisory lock is obtained on a networked filesystem such as NFS it is possible that the lock might get lost. This may happen due to administrative action on the server, or due to a network partition[174] which lasts long enough for the server to assume that the client is no longer functioning. When the filesystem determines that a lock has been lost, future $read(2)$ or $write(2)$ requests may fail with the error EIO. This error will persist until the lock is removed or the file descriptor is closed. Since Linux 3.12, this happens at least for $NFSv4$ (including all minor versions). Some versions of UNIX send a signal ($SIGLOST$) in this circumstance.[175]

---

[171] **Warning:** the Linux implementation of mandatory locking is unreliable. Because of these bugs, and the fact that the feature is believed to be little used, since Linux 4.5, mandatory locking has been made an optional feature, governed by a configuration option ($CONFIG\_MANDATORY\_FILE\_LOCKING$). This is an initial step toward removing this feature completely.

[172] POSIX does not specify Mandatory locking. Some other systems also support mandatory locking, although the details of how to enable it to vary across systems.

[173] See $chmod(1)$ and $chmod(2)$.

[174] i.e., loss of network connectivity with the server.

[175] Linux does not define this signal and does not provide any asynchronous notification of lost locks.

Mutex and recursive mutex

A mutex is a special variable that is used to synchronize threads by using an advisory locking mechanism. The mutex is a simple variable that when accessing it by "lock", it will block until no actual locks are left (by other, not by the same thread).

Question: What happens when I access a locked mutex and lock it again?
Answer: There are two ways to handle this situation:

- One method called the "**non-recursive mutex**" is just to ignore that lock attempt, and thus locking works only once and unlock will unlock the mutex regardless how many times it was locked.
- The second method is a **recursive mutex** which keeps track on how many times that mutex was locked by that thread. This allows the mutex to be locked without ever reaching a deadlock situation. When we unlock the mutex, the lock counter goes down and when it reaches 0, the mutex is fully unlocked.

A recursive mutex is used when the same thread needs to lock the same mutex number of times within nested function calls.

- POSIX doesn't define the default behavior.
- In Linux, the default behavior is recursive mutex.
- There are UNIX systems (like Solaris), where the default behavior is non-recursive.
- We mainly work on Linux and thus we start from the assumption of using recursive mutex.

$pthread\_mutex\_init(3)$ and $pthread\_mutex\_destroy(3)$

Those library functions initialize and destroy a mutex object.

From $man(3)$:

The $pthread\_mutex\_destroy(pthread\_mutex\_t\ *mutex)$ function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. An implementation may cause $pthread\_mutex\_destroy()$ to set the object referenced by mutex to an invalid value.

A destroyed mutex object can be reinitialized using $pthread\_mutex\_init(pthread\_mutex\_t\ * restrict\ mutex, const\ pthread\_mutexattr\_t\ * restrict\ attr)$; the results of otherwise referencing the object after it has been destroyed are undefined. It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a $pthread\_cond\_timedwait()$ or $pthread\_cond\_wait()$ call by another thread, results in undefined behavior.

Critical bugs initialization and destruction:

- Attempting to initialize an already initialized mutex results in undefined behavior.
- The behavior is undefined if the value specified by the mutex argument to $pthread\_mutex\_destroy()$ does not refer to an initialized mutex.
- The behavior is undefined if the value specified by the $attr$ argument to $pthread\_mutex\_init()$ does not refer to an initialized mutex attributes object.

$pthread\_mutex\_lock(3)$, $pthread\_mutex\_unlock(3)$ and $pthread\_mutex\_trylock(3)$

Those library functions are used to lock and unlock a mutex object.

From $man(3)$:

The mutex object referenced by mutex shall be locked by a call to $pthread\_mutex\_lock(pthread\_mutex\_t\ *mutex)$ that returns zero or $[EOWNERDEAD]$. If the mutex is already locked by another thread, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner. If a thread attempts to relock a mutex that it has already locked, $pthread\_mutex\_lock(pthread\_mutex\_t\ *mutex)$ shall behave as described in the Relock column of the following table

The $pthread\_mutex\_trylock(pthread\_mutex\_t\ *mutex)$ function shall be equivalent to $pthread\_mutex\_lock()$, except that if the mutex object referenced by mutex is currently locked,[176] the call shall return immediately. If the mutex type is $PTHREAD\_MUTEX\_RECURSIVE$ and the mutex is currently owned by the calling thread, the mutex lock count shall be incremented by one and the $pthread\_mutex\_trylock()$ function shall immediately return success.

The $pthread\_mutex\_unlock(pthread\_mutex\_t\ *mutex)$ function shall release the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when $pthread\_mutex\_unlock()$ is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.[177]

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.

If mutex does not refer to an initialized mutex object, the behavior of $pthread\_mutex\_lock()$, $pthread\_mutex\_trylock()$, and $pthread\_mutex\_unlock()$ is undefined.

$pthread\_mutex\_timedlock(3)$

This function locks a mutex object with a time limit.

From $man(3)$:

The $pthread\_mutex\_timedlock(pthread\_mutex\_t\ *restrict\ mutex, const\ struct\ timespec\ *restrict\ abstime)$ function shall lock the mutex object referenced by mutex. If the mutex is already locked, the calling thread shall block until the mutex becomes available as in the $pthread\_mutex\_lock()$ function. If the mutex cannot be locked without waiting for another thread to unlock the mutex, this wait shall be terminated when the specified timeout expires.

---

[176] By any thread, including the current thread.

[177] In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex shall become available when the count reaches zero and the calling thread no longer has any locks on this mutex.

The timeout shall expire when the absolute time specified by $abstime$ passes, as measured by the clock on which timeouts are based,[178] or if the absolute time specified by $abstime$ has already been passed at the time of the call.

The timeout shall be based on the $CLOCK\_REALTIME$ clock. The resolution of the timeout shall be the resolution of the clock on which it is based. The $timespec$ data type is defined in the $< time.h >$ header file.[179]

As a consequence of the priority inheritance rules,[180] if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex shall be adjusted as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

If mutex does not refer to an initialized mutex object, the behavior is undefined.

[178] That is, when the value of that clock equals or exceeds $abstime$.
[179] Under no circumstance shall the function fail with a timeout if the mutex can be locked immediately. The validity of the $abstime$ parameter need not be checked if the mutex can be locked immediately.
[180] For mutexes initialized with the $PRIO\_INHERIT$ protocol.

Condition

For many applications, mutual exclusion is not enough. Threads attempting an operation may need to wait until some condition P holds true. A busy waiting loop will not work, as mutual exclusion will prevent any other thread from entering the monitor to make the condition true. Other "solutions" exist such as having a loop that unlocks the monitor, waits a certain amount of time, locks the monitor, and checks for the condition P. Theoretically, it works and will not deadlock, but issues arise. It is hard to decide an appropriate amount of waiting time: too small and the thread will hog the CPU, too big and it will be apparently unresponsive. What is needed is a way to signal the thread when the condition P is true.

The solution is to use condition variables. Conceptually a condition variable is a queue of threads, associated with a mutex, on which a thread may wait for some condition to become true.

<u>Thus, there are three main operations on condition variables</u>:

- $wait\ c, m$ – Where c is a condition variable and m is a mutex (lock) associated with the monitor. This operation is called by a thread that needs to wait until the assertion P is true before proceeding. While the thread is waiting, it does not occupy the monitor. The function, and fundamental contract, of the "wait" operation, is to do the following steps:
    - Atomically:
        1) Release the mutex m.
        2) Move this thread from the "running" to c's "wait-queue"[181] of threads.
        3) Sleep this thread.[182]
    - Once this thread is subsequently notified/signaled and resumed, then automatically re-acquire the mutex m.
- $signal\ c$ – Also known as $notify\ c$, is called by a thread to indicate that the assertion P is true. Depending on the type and implementation of the monitor, this moves one or more threads from c's sleep-queue to the "ready queue", or another queue for it to be executed. It is usually considered a best practice to perform the "signal" operation before releasing mutex m that is associated with c, but as long as the code is properly designed for concurrency and depending on the threading implementation, it is often also acceptable to release the lock before signaling. Depending on the threading implementation, the ordering of this can have scheduling-priority ramifications. A threading implementation should document any special constraints on this ordering.
- $broadcast\ c$ – Also known as $notifyAll\ c$, is a similar operation that wakes up all threads in c's wait-queue. This empties the wait-queue. Generally, when more than one predicate condition is associated with the same condition variable, the application will require broadcast instead of signal because a thread waiting for the wrong condition might be woken up and then immediately go back to sleep without waking up a thread waiting for the correct condition that just became true. Otherwise, if the predicate condition is one-to-one with the condition variable associated with it, then signal may be more efficient than broadcast.

[181] a.k.a. "sleep-queue".
[182] Context is synchronously yielded to another thread.

$pthread\_cond\_init(3)$ and $pthread\_cond\_destroy(3)$

Those functions initialize and destroy a condition object.

From $man(3)$:

The $pthread\_cond\_destroy(pthread\_cond\_t\ *cond)$ function shall destroy the given condition variable specified by cond; the object becomes, in effect, uninitialized. An implementation may cause $pthread\_cond\_destroy()$ to set the object referenced by cond to an invalid value. A destroyed condition variable object can be reinitialized using $pthread\_cond\_init()$. It shall be safe to destroy an initialized condition variable upon which no threads are currently blocked.

The $pthread\_cond\_init(pthread\_cond\_t\ *restrict\ cond, const\ pthread\_condattr\_t\ *restrict\ attr)$ function shall initialize the condition variable referenced by cond with attributes referenced by $attr$. If $attr$ is NULL, the default condition variable attributes shall be used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable shall become initialized.

Critical bugs in initialization and destruction:

- Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.
- Attempting to initialize an already initialized condition variable results in undefined behavior.
- The results of referencing the object after it has been destroyed are undefined.
- The behavior is undefined if the value specified by the cond argument to $pthread\_cond\_destroy()$ does not refer to an initialized condition variable.
- The behavior is undefined if the value specified by the $attr$ argument to $pthread\_cond\_init()$ does not refer to an initialized condition variable attributes object.

$pthread\_cond\_signal(3)$ and $pthread\_cond\_broadcast(3)$

Those functions signal to release a conditional object.

From $man(3)$:

These functions shall unblock threads blocked on a condition variable.

The $pthread\_cond\_broadcast(pthread\_cond\_t\ *cond)$ function shall unblock all threads currently blocked on the specified condition variable cond.

The $pthread\_cond\_signal(pthread\_cond\_t\ *cond)$ function shall unblock at least one of the threads that are blocked on the specified condition variable cond.[183]

If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked. When each thread unblocked as a result of a $pthread\_cond\_broadcast()$ or $pthread\_cond\_signal()$ returns from its call to $pthread\_cond\_wait()$ or $pthread\_cond\_timedwait()$, the thread shall own the mutex

[183] If any threads are blocked on cond.

with which it called $pthread\_cond\_wait()$ or $pthread\_cond\_timedwait()$. The thread(s) that are unblocked shall contend for the mutex according to the scheduling policy,[184] and as if each had called $pthread\_mutex\_lock(3)$.

The $pthread\_cond\_broadcast()$ and $pthread\_cond\_signal()$ functions shall have no effect if there are no threads currently blocked on cond.

The behavior is undefined if the value specified by the cond argument to $pthread\_cond\_broadcast()$ or $pthread\_cond\_signal()$ does not refer to an initialized condition variable.

$pthread\_cond\_timedwait(3)$ and $pthread\_cond\_wait(3)$

Those functions wait for condition to be signaled.

From $man(3)$:

The $pthread\_cond\_timedwait(pthread\_cond\_t\ *restrict\ cond, pthread\_mutex\_t\ *restrict\ mutex, const\ struct\ timespec\ *restrict\ abstime)$ and $pthread\_cond\_wait(pthread\_cond\_t\ *restrict\ cond, pthread\_mutex\_t\ *restrict\ mutex)$ functions shall block on a condition variable. The application shall ensure that these functions are called with mutex locked by the calling thread; otherwise, an error[185] or undefined behavior[186] results.

These functions atomically release mutex and cause the calling thread to block on the condition variable cond.[187] That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to $pthread\_cond\_broadcast()$ or $pthread\_cond\_signal()$ in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread. If mutex is a robust mutex where an owner terminated while holding the lock and the state is recoverable, the mutex shall be acquired even though the function returns an error code.

The behavior is undefined if the value specified by the cond or mutex argument to these functions does not refer to an initialized condition variable or an initialized mutex object, respectively.

[184] If applicable.
[185] For $PTHREAD\_MUTEX\_ERRORCHECK$ and robust mutexes.
[186] For other mutexes.
[187] Atomically here means “atomically with respect to access by another thread to the mutex and then the condition variable”.

Lecture 7

Introduction to design patterns

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.[188]

Singleton[189]

The singleton design patterns defines that an instance of an object can be created only once. The difference between singleton object and global object is that the singleton object is thread-safe using mutex locking, to ensure that only one thread can create the object. The singleton pattern can also be used as a basis for other design patterns, such as the abstract factory, factory method, builder, and prototype patterns. Façade objects are also often singletons because only one façade object is required.

Logging is a common real-world use case for singletons because all objects that wish to log messages require a uniform point of access and conceptually write to a single source.

Factory Method[190]

The Factory Method design pattern is a creational design pattern that provides an interface for creating objects in a super class but allows sub-classes to change the type of objects that are created. This is achieved by configuring the method that is used to create object, which is often called "The Factory Method". The sub-classes can override this method to change the class of the objects that would be created.

The Factory Method pattern is used when a class can't predict the type of object it needs to create and wants to allow her sub-classes to point those objects. It's a way of moving the implementation process to the sub-classes, which allows free integration and reuse of code.

The pattern contains the following parts:

- **Product** – This is the interface of the type of objects that the pattern creates.
- **Specific Product** – This is the object that implements the followed interface.
- **Producer** – An abstract class which contains the relevant signatures. The producer returns an object with the type of the product.
- **Specific Producer** – A class that extends the producer and replaces the design pattern to an instance of a specific product.

---

[188] In this lecture we will introduce few design patterns, as they appear in the book Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects, by Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, and the book Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
[189] This design pattern is learned on the Object-oriented Programming course, and thus is less relevant to this course.
[190] This design pattern is learned on the Object-oriented Programming course, and thus is less relevant to this course.

We can look at this design pattern as a hash map[191] that maps between a key[192] and his constructor.[193]

An example for usage of this design pattern is the $socket(2)$ system call.

Adapter[194]

This design pattern acts as a bridge between two incompletable interfaces. It is used to adapt one interface to another, even if they are completely different.

For example: If we have a thread that waits to **cond** and when the **cond** arrives, it sends a message to a socket, to allow us to do the $select(2)$ system call on the **cond**.

Façade

This is the more efficient design pattern to manage and simplify complex software system. This is a wrapper class which purpose is to take a complex system and wrap it in a simple and understandable API. It's easy to monitor how a more complex system works using this design pattern.

For example: The file descriptors are using Façade because its functions work for files, sockets, etc.

Reactor

The Reactor design pattern is a behavioral design pattern that is designed to be used by systems that are triggered via events. Mainly used by applications that need to support multiple service requests from multiple different clients, in parallel.

The main idea of this pattern is to handle different service requests that are transferred to the application. Each application may use a different method to handle the request and the applications don't know which requests will be received and when. This pattern allows effective timing of those services by demultiplexing the incoming requests and handling only the "hot" requests by a single thread sequentially.[195]

The pattern contains the following parts:

- **Reactor** – A scheduler that takes care of the demultiplexing the incoming requests and sending each one to the correct event handler.
- **Synchronization Event Demultiplexer** – This part listens to the sources for possible incoming event and waits until one of them will “wake up” and be active.[196] When

[191] Or other mapping tool.
[192] For example, a socket type.
[193] For example, building the right socket.
[194] This design pattern is learned on the Object-oriented Programming course, and thus is less relevant to this course.
[195] For example, by the $select(2)$ system call.
[196] Send a request.

one of them indeed wakes up, it passes the request to the reactor for processing and sending it to the correct event handler.[197]

- **Event Handler** – Those are the objects that are adjusted for the specific application, that are attached to specific events. They process those events that the reactor passes to them.

We actually can see that the reactor design pattern handles the requests sequentially, in a single thread.[198] It could be that other patterns are handling each request in a new thread or a group of threads that handles them.[199]

Reactor is optimized for fast handling of each request.

Active Object

The Active Object design pattern represents a thread with a queue of tasks to be performed. The thread exists in a busy loop that checks the queue of tasks, each time dequeuing the task and handling it.

Pipeline

The Pipeline design pattern organizes a system to a series of step, that each one is performed by a single active object, that processes the data and delivers the processed data to the next active object. The delivery process is done using a pipeline – An output of one's step becomes an input of other's step. Each step has a well predefined task and each one works without a dependency from other steps.

Example for using a pipeline design pattern:

A system on the internet that receives encrypted requests. Each request is going through those five steps:

- **Thread #1** – Decodes the algorithm that's used for encryption.
- **Thread #2** – Decrypts the request using the decoded algorithm.
- **Thread #3** – Answers the request.
- **Thread #4** – Building the answer packet.
- **Thread #5** – Encrypting the answer packet and sending it back.

[197] In fact, this part is implemented using the $select(2)$ or $poll(2)$ system calls APIs.
[198] Same as the examples in Beej's guide to networking of the $select(2)$ and $poll(2)$ system calls.
[199] This is out of scope for this course.

Lecture 8

Kernel

Question: What is an operating system[200]?
Answer: An operating system is a computer system that mediate between the computer's hardware and processes that run. We need a minimal set of tools to do this mediate.

The kernel is a software that mediate between the hardware of the machine and the running processes in the machine.[201]

When the operating systems began to appear, one of the most noticeable was about the size of the kernel. There are mainly two types of kernels:

- **Micro-Kernel (Small Kernel)** – A kernel that contains only the minimum functionality as possible, and all the other stuff will be done on external processes. Enhance security and decreases the number of bugs in the kernel and moves them more to the user-space. Bugs that occur in the user-space are less critical and thus the security is less affected. It's also easier to find and solve bugs in the user-space.
- **Monolithic Kernel (Big Kernel)** – A big kernel has the advantage of performance. The context switching between the user-space and kernel-space is very expensive, and they occur rapidly in Micro-Kernel. In monolithic kernel, all the operations are done in the kernel itself without any intervention from external processes and without any context switching. The huge disadvantage of this is that the kernel is more pronounced to bugs. A bug in one of the drivers could make the whole system to fail and crash. When context switching does occur, in big kernels it's even more expensive on resources.

Linux started directly from monolithic kernel that supported all the hardware that was available at that time. The Linux kernel was very big, but also supported modules and real-time load and unload modules, to decrease the size of the kernel in runtime.

Windows kernel started as a micro-kernel (DOS core) and gradually Microsoft increased it more and more to improve performance.[202]

Today, each operating system has a Hybrid Kernel, that allows adding and removing modules from the kernel in runtime and thus increase performance or increase stability as needed.

In this course we'll learn how to write kernel modules for Linux.

[200] We talk about the technological aspect and not the business aspect. For example, Windows was sold with Windows Explorer, Windows Media Player, and FreeCell. It's obvious that we don't talk about FreeCell when we talk about an operating system.
[201] This is in a nutshell, because we don't want to enter this realm too much.
[202] For example, moving the GPU completely to the kernel-space for increasing performance in video games.

Linux kernel modules[203]

A simple example of kernel module would be something like this:[204]

```
/*
 * hello-1.c - The simplest kernel module.
*/
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

int init_module(void) {
        pr_info("Hello world 1.\n");

        /* A non 0 return means init_module failed; module can't be loaded. */
        return 0;
}

void cleanup_module(void) {
        pr_info("Goodbye world 1.\n");
}

MODULE_LICENSE("GPL");
```

We will immediately notice few changes in working with kernel modules comparing to user-space, where we worked until now.

The main difference lays on the fact that the kernel replays to requests and doesn't run a specific program. Another key difference is that in user-space we use a lot of system calls that they in fact directed to the kernel itself, which is no longer possible as we work in the kernel-space now.

<u>Kernels-space functions explanation</u>:

- $init\_module()$ – This is the starting point of the module when it's loaded to the kernel, in a nutshell, similar to $main()$ in user-space, although it's not true, as the module provides a service and does not run the main function and exits. The counterpart in Windows is $dllmain()$ and in GCC is $attribute(constructor)$.
- $cleanup\_module()$ – The ending point of the module when it's unloaded from the kernel. In this function we should write all the cleanup commands to release any resources that were occupied by the module while it was loaded.[205]
- $pr\_info()$ – Replaces the $printf(3)$ function, as it's a library function and in kernel space we don't have access to any libraries, even the standard C library. The $pr\_info()$ function prints the message to some location (it could be a log file, if there is any).[206]
- $MODULE\_LICENSE()$ – Since the Linux kernel was published under the GNU General Public License (GNU GPL) 2.0, any Linux kernel module should be also published under free software license.

---

[203] For the following lectures we will work with The Linux Kernel Module Programming Guide by Peter Jay Salzman, Michael Burian, Ori Pomerantz, Bob Mottram, and Jim Huang. Online copy is **avaliable here**. It's recommended to try to compile yourself all the code examples (Chapter 1–6 for this lecture).
[204] This kernel module practically does nothing and only print to the kernel log (if there is any) about the load and unload of this module.
[205] Between the $init\_module()$ and the $cleanup\_module()$, the actual module service code is running.
[206] The $pr\_info()$ uses the $printk()$ which is a special kernel function.

We also can notice that the include header files are that of the kernel itself, and not the standard C library headers.[207]

Since regular compiling with GCC[208] isn't relevant anymore, because it searches header files that aren't relevant for kernel modules, we'll use a special version of makefile:[209]

```
obj-m += hello-1.o
PWD := $(CURDIR)

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

When we want to add a kernel module to the compiling list, we add it with via $obj - m$ variable. In this example, when we'll run the $make$ command, it'll compile a kernel module named "hello-1.ko". To insert the module, we must have root privileges or use the $sudo$ command.

```
modinfo hello-1.ko
```

At this point the command:

```
sudo lsmod | grep hello
```

should return nothing. You can try loading your shiny new module with:

```
sudo insmod hello-1.ko
```

The dash character will get converted to an underscore, so when you again try:

```
sudo lsmod | grep hello
```

you should now see your loaded module. It can be removed again with:

```
sudo rmmod hello_1
```

Notice that the dash was replaced by an underscore. To see what just happened in the logs:

```
sudo journalctl --since "1 hour ago" | grep kernel
```

There are at least thousands of kernel modules in the kernel itself. To manage the modules effectively, each module must have a unique name to distinguish it from others. Examples would be to distinguish it from other modules in the kernel log output, to check for errors.

[207] The prefix $linux/$ in the include header file suggests that it's related to the kernel.
[208] GNU C Compiler
[209] If you use smart IDEs to code your kernel module like Visual Studio Code, it might fail to recognize the header files. In this case, you'll need to reconfigure the IDE to recognize those header files, or to ignore those errors.

We can also use macros for the load and unload functions of the kernel module:

```
/*
 * hello-2.c - Demonstrating the module_init() and module_exit() macros.
 * This is preferred overusing init_module() and cleanup_module().
*/
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

static int __init hello_2_init(void) {
        pr_info("Hello, world 2\n");
        return 0;
}

static void __exit hello_2_exit(void) {
        pr_info("Goodbye, world 2\n");
}

module_init(hello_2_init);
module_exit(hello_2_exit);

MODULE_LICENSE("GPL");
```

$module\_init()$ and $module\_exit()$ are two macros that create an initialize and clean functions without explicitly declaring them and allowing them to be named differently.

To reduce the kernel size, we could use the $module\_init()$ macro to create a temporary initialize function, which will be released once the module is fully loaded to the kernel. We could also use the $module\_exit()$ macro to removing the need of the cleanup function to enter the memory of the kernel when doing context switching.

The $\_\_initdata$ macro allows use to allocate a temporary memory for a variable[210] in the initialize process, and the memory will be released once the initialize will be done:

```
/*
 * hello-3.c - Illustrating the __init, __initdata and __exit macros.
*/
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

static int hello3_data __initdata = 3;

static int __init hello_3_init(void) {
        pr_info("Hello, world %d\n", hello3_data);
        return 0;
}

static void __exit hello_3_exit(void) {
        pr_info("Goodbye, world 3\n");
}

module_init(hello_3_init);
module_exit(hello_3_exit);

MODULE_LICENSE("GPL");
```

[210] Not an array.

In tutorial 4 of the guide, they show an example of inserting extra infromation to the module, which can be viewed via the CMD tool command $modinfo$:

```
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("LKMPG");
MODULE_DESCRIPTION("A sample driver");
static int __init init_hello_4(void) {
        pr_info("Hello, world 4\n");
        return 0;
}
static void __exit cleanup_hello_4(void) {
        pr_info("Goodbye, world 4\n");
}
module_init(init_hello_4);
module_exit(cleanup_hello_4);
```

In tutorial 5 of the guide, they show an example of adding arguments when loading a module to the kernel. For each parameter, we can add a description:

```
static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";
static int myintarray[2] = { 420, 420 };
static int arr_argc = 0;
module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");
module_param_array(myintarray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintarray, "An array of integers");
```

We can pop the information for each parameter by using the following command:

$$\$modinfo\ < module\ name.ko >$$

Note that we can build one kernel module from a few C source files. For example, from those C source files called $start.c$ and $stop.c$, we can build a kernel module named $startstop$ using the following line in makefile:

$$startstop - objs := start.o\ stop.o$$

```c
/*
 * start.c - Illustration of multi filed modules
 */

#include <linux/kernel.h> /* We are doing kernel work */
#include <linux/module.h> /* Specifically, a module */

int init_module(void)
{
    pr_info("Hello, world - this is the kernel speaking\n");
    return 0;
}

MODULE_LICENSE("GPL");
```

```c
/*
 * stop.c - Illustration of multi filed modules
 */

#include <linux/kernel.h> /* We are doing kernel work */
#include <linux/module.h> /* Specifically, a module  */

void cleanup_module(void)
{
    pr_info("Short is the life of a kernel module\n");
}

MODULE_LICENSE("GPL");
```

Character device driver

We can build more interactive kernel modules, with the ability to respond to user commands.

In UNIX we work with using files. When the user "polls" a specific file he is actually requesting from the kernel, using specific commands. There are two types of drivers in the Linux kernel, that work almost the same: Character Device and Block Device.

We'll try to simplify the definition:

A character device works a character by character,[211] while a block device works with the block method. This definition is far cry from the truth. Another simplified definition is when we try to search in the device itself, if it's possible then it's a block device, otherwise it's a character device.

The most accurate definition is:[212]

- **Block Layer** – A layer in the system that manages the input and output.
- **Block Device** – A device that uses the block layer to manage the requests. The block layer can change the order of the requests, change the requests themself for optimization purposes, etc. A block device is harder to debug, and thus we'll start with a character device.
- **Character Device** – A device that doesn't use the block layer, and he manages the requests himself.

The definition of the $file\_operations$ struct is defined in $linux/fs.h$ header file. This struct contains pointers to functions that were defined by the driver itself and are allowing to do operations on the driver. Each field in the struct matches a memory address of a function, that was defined by the driver to handle the specific request. For example, every character manager needs to implement a function that represents reading from the device. The $file\_operations$ struct contains the address of the function of the module that handles the request.

The definition for Linux kernel version 5.4:

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iopoll)(struct kiocb *kiocb, bool spin);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
```

[211] Byte by byte.
[212] It's worth to note that in Linux everything is a file and thus we work mainly with files.

Not all the functions are required to be implemented. For example, the driver for the GPU shouldn't read from a directory struct. Any action we don't want to implement we must set the function pointer to NULL.

For example:

```
struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

To create a new device driver, we can use the $register\_chrdev()$ function:

$$int\ register\_chrdev(unsigned\ int\ major, const\ char\ *name, struct\ file\_operations\ *fops);$$

To delete a character device, we can use the unregister function.

Using the major and minor numbers, we can create some devices from the same type, and still distinguish between them. The $device\_create()$ function associates a component to the file. The $exit()$ function disables the component and releases the character device.

Example:[213]

```
static int __init chardev_init(void)
{
    major = register_chrdev(0, DEVICE_NAME, &chardev_fops);

    if (major < 0) {
        pr_alert("Registering char device failed with %d\n", major);
        return major;
    }

    pr_info("I was assigned major number %d.\n", major);

    cls = class_create(THIS_MODULE, DEVICE_NAME);
    device_create(cls, NULL, MKDEV(major, 0), NULL, DEVICE_NAME);

    pr_info("Device created on /dev/%s\n", DEVICE_NAME);

    return SUCCESS;
}
```

To get in touch with the user-space, we can use the $get\_user()$ and $put\_user()$ functions to copy the memory from the kernel-space to the user-space and vice versa. We can also use the $copy\_user()$ function in the same way.

We can also create a character device to communicate with the kernel using the CMD tool command $mknod(1)$.

[213] The full code is avaliable at the guide, chapter 6.4. It is recommended to copy the code, compile, and understand the commands.

Lecture 9

Character devices

Reminder from the previous lecture:

In the "file tree" of the system, files that aren't directories are splatted between character device and block device.

- **Block Layer** – A layer in the operating system that manages the input and output (IO), where the kernel does optimization and scheduling to requests.
  It can evade by balancing the requests. **For example**, program A sends 10 requests and then program B requests, then a balance will be made between the requests rate of A and B.
  It can also be expressed in continuity and efficiency. **For example**, program A requests resources 1 and 2, program B requests resource 1000 and then program A requests resource 4, then it's easier to read continuously, so resources 1 to 4 will return[214] and then resources 1000 will return to program B.
- **Block Device** – A device managed by the block layer.[215]

We will now continue the discussion and focus on character devices.

Character Device – A device that doesn't undergo optimization.[216] Using this kind of device, we can easily communicate with the kernel and execute all kinds of functions. For example, creating character device that we'll read from it and write to it. [217]

Examples for character devices: mouse, keyboard, etc.

Each character device has a major number and a minor number:

- **Major Number** – An identification number of the type of the driver that is assigned to handle a certain device.[218] Why do we need it? There could be a lot of character devices in the system,[219] and thus we'll want that each one will get a unique id.
- **Minor Number** – A minor number is the serial number for that device. It's relevant when there are at least two devices of the same type.[220]

To create a new character device, we will use the following function:

$$int\ register\_chrdev(unsigned\ int\ major, const\ char\ *name, struct\ file\_operations\ *fops);$$

[214] Resource 3 will drop out because it's easier.
[215] Because the block layer sometimes returns stuff we didn't request and also not on the order we wanted, it's complicated. As a result, we'll focus now on character devices only.
[216] An intuitive definition from lecture 8.
[217] And thus, we'll write and read from the RAM indirectly.
[218] For example, a driver to the keyboard, a driver to the mouse, when everyone will have a separated major number.
[219] Even two of the same type, like for example, two mice.
[220] For example, if we have 4 joysticks, their major number will be the same, but everyone will get a different minor number, so the driver could distinguish between them.

Arguments breakdown:

- $major$ – The major number. The user can put a specific major number or put 0 for the system to assign automatically.
- $name$ – The name of the device that will appear in $/proc/devices$.
- $fops$ – A bit map to all the operation functions that can be called to the device.

Example of an initialization function for initializing and create a character device:

```
static int __init chardev_init(void) {
        major = register_chrdev(0, DEVICE_NAME, &chardev_fops);

        if (major < 0)
        {
                pr_alert("Registering char device failed with %d\n", major);
                return major;
        }

        pr_info("I was assigned major number %d.\n", major);
        cls = class_create(THIS_MODULE, DEVICE_NAME);
        device create(cls, NULL, MKDEV(major, 0), NULL, DEVICE_NAME);

        pr_info("Device created on /dev/%s\n", DEVICE_NAME);

        return SUCCESS;
}
```

If the $register\_chrdev()$ function returns a negative value, it means that it failed to register the character device. Otherwise, it succeeds, and we'll use the $device\_create()$ to create the device itself in $/dev$. After the device is created, we can write and read from it.

Example of cleanup and exit function of character device:

```
static void __exit chardev_exit(void) {
        device_destroy(cls, MKDEV(major, 0));
        class_destroy(cls);

        /* Unregister the device */
        unregister_chrdev(major, DEVICE_NAME);
}
```

We destroy the device using the major number with the $device\_destory()$ function. After that, we'll use the $unregister\_chrdev()$ function to remove the character device completely and disallowing the programmers to create those type of devices, as the functions that handle it are no longer recognized.

Example of a device open function implementation when the user uses the $open(2)$ system call:

```
static int device_open(struct inode *inode, struct file *file) {
        static int counter = 0;

        if (atomic_cmpxchg(&already_open, CDEV_NOT_USED, CDEV_EXCLUSIVE_OPEN))
                return -EBUSY;

        sprintf(msg, "I already told you %d times Hello world!\n", counter++);

        try_module_get(THIS_MODULE);

        return SUCCESS;
}
```

Each time we open the device, we'll write to a global buffer named msg, the message that was mentioned above.

Example of an implementation of device read function:

```
static ssize_t device_read(struct file *filp, char __user *buffer, size_t length, loff_t *offset) {
        int bytes_read = 0;
        const char *msg_ptr = msg;

        if (!*(msg_ptr + *offset))
        {
                *offset = 0;
                return 0;
        }

        msg_ptr += *offset;

        while (length && *msg_ptr)
        {
                put_user(*(msg_ptr++), buffer++);
                length--;
                bytes_read++;
        }

        *offset += bytes_read;
        return bytes_read;
}
```

The copying of the message is from the msg global buffer to the buffer that the user gave us. Because the copying operation is done from the kernel to the user, we need to user the macros $get\_user()$ and $put\_user()$ that are copying data from the user-space to the kernel-space and vice versa. If we want to copy a buffer, we can use the $copy\_to\_user()$ or $copy\_from\_user()$ functions. Pay attention that the while loop will end when the buffer ends or the current message ends, meaning the length reaches 0 or the pointer of the message reaches 0.

I/O control[221]

This this chapter we'll be focused on adding to the character device 2 features:

- **Writing** – Along with the reading from the device, we want to add the ability to write to the device.
- **A Joker Function** – We would like a function that will allow us to perform on the device operations that are not customary and conventional to perform on a file.[222] For example: If we have a tape machine where there is a robot that knows how to read and write tapes, if we'll them him to replace a tape, he won't know what to do.

The solution: the $ioctl(2)$ system call.

IOCTL – The $ioctl(2)$[223] system call is a function that recives a file descriptor that we are working with and a number of actions that only the specific file recognizes,[224] and able to do the action on it. The function also allows to pass parameters for the action.[225]

Example of usage: Suppose we have a TV card. With the $open(2)$ system call we'll open it, with the $read(2)$ system call we'll read the frames it captures, with the close(2) system call we'll close it, and now with the $ioctl(2)$ system call we could also change a channel, change the resolution, change to volume, and even change the codec. Each one of the $ioctl$s will receive different parameters.[226]

Note: Because most of the functions are similar to the previous example[227] we will only show the interesting functions.

The file operations structure:

```
static struct file_operations fops = {
        .owner = THIS_MODULE,
        .open = test_ioctl_open,
        .release = test_ioctl_close,
        .read = test_ioctl_read,
        .unlocked_ioctl = test_ioctl_ioctl,
};
```

Now we'll add to our function collection the $test\_ioctl\_ioctl()$ function, and this function will be executed when we call the $ioctl(2)$ system call.[228]

[221] Chapter 9 in the Linux Kernel Module Programming Guide, click here.
[222] Functions that are different from reading , writing, etc.
[223] Acronyms for **I**nput **O**utput **C**on**T**ro**L**.
[224] For example, in a mouse it could be to adapt it to lefties, in TV card it could be changing a channel, etc.
[225] For example, a quality flag that can get the values low, medium, or high.
[226] For example, one w will get the channel number, the other will get the codec.
[227] Like the initialize function, the cleanup function and the read and write functions.
[228] We'll show the implementation in the next page.

The $test\_ioctl\_ioctl()$ function implementation:

```
static long test_ioctl_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {
	struct test_ioctl_data *ioctl_data = filp->private_data;
	unsigned char vale;
	struct ioctl_arg data;
	memset(&data, 0, sizeof(data));

	switch (cmd)
	{
		case IOCTL_VALSET:
		{
			if (copy_from_user(&data, (int __user *)arg, sizeof(data)))
				return -EFAULT;

			pr_alert("IOCTL set val:%x .\n", data.val);

			write_lock(&ioctl_data->lock);
			ioctl_data->val = data.val;
			write_unlock(&ioctl_data->lock);

			return 0;
		}

		case IOCTL_VALGET:
		{
			read_lock(&ioctl_data->lock);
			val = ioctl_data->val;
			read_unlock(&ioctl_data->lock);

			data.val = val;

			if (copy_to_user((int __user *)arg, &data, sizeof(data)))
				return -EFAULT;

			return 0;
		}

		case IOCTL_VALGET_NUM:
			return __put_user(ioctl_num, (int __user *)arg);

		case IOCTL_VALSET_NUM:
		{
			ioctl_num = arg;
			return 0;
		}

		default:
			return -ENOTTY;
	}
}
```

In fact, our jocker function receives a command (cmd) and argument (arg). We do a switch case on the command, which determines what will happen.

Question: What will happen if we try to do an $ioctl(2)$ system call at the same time?
Answer: For this occasion, we defined a variable named $lock$ from type $rwlock\_t$, meaning read-write protection, and in fact is the equivalent to mutex in the kernels-space. If we would do a lock on writing, nobody else would be able to enter the function. If we do a lock on reading, others will be able to read, but won't be able to write while we read the data.

$ioctl(2)$

From $man(2)$:

The $ioctl(int\ fd, unsigned\ long\ request, \ldots)$ system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files[229] may be controlled with $ioctl()$ requests. The argument fd must be an open file descriptor.

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally $char\ *argp$,[230] and will be so named for this discussion.

An $ioctl()$ request has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument argp in bytes. Macros and defines used in specifying an $ioctl()$ request are located in the file $< sys/ioctl.h >$.

Usually, on success zero is returned. A few $ioctl()$ requests use the return value as an output parameter and return a nonnegative value on success. On error, -1 is returned, and errno is set to indicate the error.

Arguments, returns, and semantics of $ioctl()$ vary according to the device driver in question.[231]

In order to use this call, one needs an open file descriptor. Often the $open(2)$ call has unwanted side effects, that can be avoided under Linux by giving it the $O\_NONBLOCK$ flag.

$ioctl$ structure

$ioctl$ command values are 32-bit constants. In principle these constants are completely arbitrary, but people have tried to build some structure into them.

The macros describing this structure live in $< asm/ioctl.h >$ and are $\_IO(type, nr)$ and $\{\_IOR, \_IOW, \_IOWR\}(type, nr, size)$. They use $sizeof(size)$ so that size is a misnomer here: this third argument is a data type.

[229] e.g., terminals.
[230] From the days before $void\ *$ was valid C.
[231] The call is used as a catch-all for operations that don't cleanly fit the UNIX stream I/O model.

Lecture 10

File system[232]

Question: What's a file system?
Answer: Suppose we have two programs that runs on the same machine, and we want to manage their files. What characterizes each file in our hard drive? By the path.

The key is our full path of the file's location. We build a systematic hierarchy of directories, and by the name of the file we'll find its data. In our hard drive there are a lot of blocks (blocks of memory), and as we said, part of the blocks contains the data of the specific file, and another part contains a data of a different file.

Question: How we'll know what's the data of the file?
Answer: In each file we have a data frame, for example what's the size of the file, what are the access permissions to it, etc. All this information is saved on a special sector called "inode".[233] In this sector, we hold a list of all the blocks that contains the data of the file, and thus we'll have a list of pointers, so we can seek a specific location in memory.

Question: What will happen if the inode memory will fill up?
Answer: We can hold a list of pointers, such that each pointer points to another list of pointers, and we can chain them together and get access to bigger files.

Question: Suppose we want to increase the size of a specific file, how we'll find the next block that's avaliable that can be used to store the enlarged file data?
Answer: We'll hold a special block called a super block, that will hold general information about the file system itself. In particular, part of the information that the super block stores are a bitmap of all the avaliable blocks.

Question: How we'll split the hard drive?
Answer: Suppose that the super block is the first block in the hard drive. After that, 10% of the hard drive would be inode blocks, and the rest 90% of the hard drive would be the data block.

Question: What will happen if we want to create a new file?
Answer: We'll find in the file system an available block and it will write the data to it.

Question: What if we have two pointers to the same inode in different directories?
Answer: First we'll notice that inode represents a file, and inode is a collection of blocks, meaning that the file is counted twice. Suppose that we add data to the file, it'll be added to two different places, and this is the way to share a file between two different users. In the past it was called a “hard link”, and it made a lot of problems.

The current solution to the problem in Windows is a shortcut (called a symbolic link) that is defined in his type that it’s a shortcut, and it contains only the path to the original file.

[232] In this lecture and the following ones, we will learn how to build a file system, and how the Unix File System (UFS) was built. Relevant chapters 3–4 from the book “Advanced Programming in the Unix Environment” BY William Richard Stevens.
[233] We can assume that this sector is equal to a block. although it's not accurate.

Question: How we can we make the files more continuous to get the best performance in low cost?
Answer: We need to do a defragmentation to the files. This is an offline operation, that takes fragmented files, which are scattered across the hard drive on non-continuous blocks, finds a section of available blocks, and moves all its the content of the file to there. The old blocks are now avaliable.

Question: How can we write in a continuous way?
Answer: We can hold blocks that have a larger capacity, but it can be wasteful on small files.

Question: How can we solve this wastefulness?
Answer: We defined a block that can contain at most a data of one file only. We can define the last block as a fragment and there we can store smaller files on the same block. When the file grows larger than the block can hold, we'll move it to a new block.

In all the issues that were mentioned above, the Unix File System (UFS) provides a solution, and modern Linux machines still support this old file system.

On solid-state drives (SSDs), we got a problem. The cells that represent the hard drive, can't be written unlimited times, as there is a wear on the cells themself. If we write and delete too much, the SSD will fail, and all the data will be lost. If our writes are distributed equally, the SSD will survive longer.

Another solution to the problem is to split the super block in more than one place in the SSD.[234] We'll hold few copies of the super block and will use only the most recently changed super block, and thus the number of write operations to the same place decreases dramatically.

[234] Not necessarily that all of them will be located in the same place in memory.

Lecture 11

Unix File System

Reminder from the previous lecture:

We dealt with how to build the Unix File System (UFS). We will repeat the material.

A disk is divided into two types:

- **Magnetic Disk** – Was in frequent use in the past. Uses an old technology of magnetic storage. For example, phonograph. Slower compering to the main memory. Usually has large capacity and is faster in sequential reading and writing but is very slow on random reads or writes.
- **SSD (Solid-State Disk)** – Newer and came into frequent use in recent years. Considered a reliable electronic media storage. compering to the main memory, but faster than a magnetic disk. Very fast on random reads and writes but has almost the same speed in sequential reading and writing.

The disks are divided into two categories:

- **Logical Disk** – Logical division of an existing disk on the computer into drives.[235]
- **Physical Disk** – A physical disk like an HDD or an SSD.

Logical disks can be made from several physical disks, thus improving reliability and performance for reading/writing.[236]

In this course, physical and logical disk are the same and we will not refer to virtualization and partitions.

File System is a place where we save files in the hard drive (locally).[237]

Question: How do we'll configure the file system?
Answer: In the Unix File System, we have huge char pointer, which we divide in a way such that we know what's available and what's occupied.

Unix File System needs to handle couple of tasks:

1. **Mapping** – Given a key (path), the UFS must return the file that's in the given path.
   Problem: Let's say for convenience that the hard drive is split into block, where each block is responsible for one file only. Suppose that A has a file that occupies 10 block and B occupies the block after A's block. What will happen when A increases the file size, as it can't override B's data.
   Possible solution: In each new write of A, we will move the contents of B.
   Actual solution: We will allow the files to be distributed non-sequentially. We'll assign special blocks that are called "Inodes" which each one will contain information about a file, among others, the pointers to the blocks that the file is stored at. The list doesn't have to be sorted by addresses.

[235] For example, drive C:\ and drive D:\.
[236] For example, we have two physical disks, and we write everything to both together, so the chance of a malfunction is significantly less because only if both are destroyed will the information be lost. In addition, the read/write rate will improve if they do not interfere with each other.
[237] There are network file systems, which are out of scope for this course.

2. **Maximum Size** – What's the maximum size of each file in the system, and how will we divide it accordingly?
   Problem: Because inode is a block, suppose that its size is k and suppose that we can have up to int (32 bits) of block. In each block we can put $\frac{32k}{4} = 8k$ pointers. Because each pointer points to a block sized k, the maximum file size is $1MB$, which of course isn't enough.
   Solution: We'll use double and triple pointers.[238]

3. **Adding to the system** – Create new inode or increasing the size of an existing file in the system.
   Problem: We want to find an avaliable block where can write data to it.
   Solution: We'll create a super-block that will be at the start of the file system and will contain a bitmap of all the available blocks.

And thus, in the Unix File System there are three main entities: **Super-block**, **Inode**, **Block** and **Fragment**.

Even though an inode contains a lot of information about the file itself, like access privileges, who is the owner, soft links, etc., it doesn't contain the name of the file, as the name of the file is an attribute of the directory that contains the file.

Whenever we connect a removeable media to the computer, there are two system calls that we need to know: $mount(2)$ and $umount(2)$.

UFS history

UFS is the modern name of a family of file systems. In any modern UNIX system, there is a file system that's based on UFS. Most modern implementations of UFS introduce new features and additional optimizations.[239] Thus, it's recommended to read the original implementation, to understand it better.

In this course we'll ignore the decisions made about where to store the information on in disk.

[238] Block that contains pointer to other blocks that contain pointers to other blocks that contain the file data itself.
[239] For example, the classic UNIX system can be coded in 1,000 lines of code at most (**GitHub example**). A modern UNIX system uses the $ext4$ file system, which was coded in 60,000 lines of code (meaning, 59,000 lines of code are done for optimization purposes).

$mount(2)$

A system call that helps to "ride" the file system and put it to use.

<u>From $man(2)$</u>:

$mount(const\ char\ *source, const\ char\ *target, const\ char\ *filesystemtype, unsigned\ long\ mountflags, const\ void\ *data)$ attaches the filesystem specified by $source$[240] to the location[241] specified by the $pathname$ in target.

Appropriate privilege[242] is required to mount filesystems.

Values for the $filesystemtype$ argument supported by the kernel are listed in $/proc/filesystems$.[243] Further types may become available when the appropriate modules are loaded.

The data argument is interpreted by the different filesystems. Typically, it is a string of comma-separated options understood by this filesystem.[244] This argument may be specified as NULL, if there are no options.

A call to $mount()$ performs one of a number of general types of operation, depending on the bits specified in $mountflags$. The choice of which operation to perform is determined by testing the Bits set in $mountflags$, with the tests being conducted in the order listed here:

- Remount an existing mount: $mountflags$ includes $MS\_REMOUNT$.
- Create a bind mount: $mountflags$ includes MS_BIND.
- Change the propagation type of an existing mount: $mountflags$ includes one of MS_SHARED, $MS\_PRIVATE$, MS_SLAVE, or $MS\_UNBINDABLE$.
- Move an existing mount to a new location: $mountflags$ includes MS_MOVE.
- Create a new mount: $mountflags$ includes none of the above flags.

Since Linux 2.4 a single filesystem can be mounted at multiple mount points, and multiple mounts can be stacked on the same mount point.

---

[240] Which is often a $pathname$ referring to a device but can also be the $pathname$ of a directory or file, or a dummy string.
[241] A directory or file.
[242] **<u>Linux</u>:** the CAP_SYS_ADMIN capability.
[243] e.g., "$btrfs$", "$ext4$", "jfs", "xfs", "$vfat$", "$fuse$", "$tmpfs$", "$cgroup$", "proc", "$mqueue$", "nfs", "$cifs$", "$iso9660$".
[244] See $mount(8)$ for details of the options available for each filesystem type.

$umount(2)$

A system call that helps to "ride" the file system and remove it from the system.

From $man(2)$:

$umount(const\ char\ *target)$ removes the attachment of the (topmost) filesystem mounted on target.

Appropriate privilege[245] is required to unmount filesystems.

The error values given below result from filesystem type independent errors. Each filesystem type may have its own special errors and its own special behavior:

- $EBUSY$ – $target$ could not be unmounted because it is busy.
- $EFAULT$ – $target$ points outside the user address space.
- $EINVAL$ – $target$ is not a mount point.
- $EINVAL$ – $target$ is locked.[246]
- $ENAMETOOLONG$ – A $pathname$ was longer than $MAXPATHLEN$.
- $ENOENT$ – A $pathname$ was empty or had a nonexistent component.
- $ENOMEM$ – The kernel could not allocate a free page to copy filenames or data into.
- $EPERM$ – The caller does not have the required privileges.

$umount(2)$ and shared mounts:

Shared mounts cause any mount activity on a mount, including $umount()$ operations, to be forwarded to every shared mount in the peer group and every slave mount of that peer group. This means that $umount()$ of any peer in a set of shared mounts will cause all of its peers to be unmounted and all of their slaves to be unmounted as well.

This propagation of unmount activity can be particularly surprising on systems where every mount is shared by default. On such systems, recursively bind mounting the root directory of the filesystem onto a subdirectory and then later unmounting that subdirectory with MNT_DETACH will cause every mount in the mount namespace to be lazily unmounted.

To ensure $umount()$ does not propagate in this fashion, the mount may be remounted using a $mount(2)$ call with a $mount\_flags$ argument that includes both MS_REC and $MS\_PRIVATE$ prior to $umount()$ being called.

Historical details:

The original $umount()$ function was called as $umount(device)$ and would return $ENOTBLK$ when called with something other than a block device. In Linux 0.98p4, a call $umount(dir)$ was added, in order to support anonymous devices. In Linux 2.3.99-pre7, the call $umount(device)$ was removed, leaving only $umount(dir)$.[247]

---

[245] **Linux:** the CAP_SYS_ADMIN capability.
[246] See $mount\_namespaces(7)$.
[247] Since now devices can be mounted in more than one place, so specifying the device does not suffice.

Blocks

Block – A place where we store data on the disk. Each block is identified using a unique address. There isn't sophistication in the way a block works. Files will often be stored in a number of discrete blocks.

Over time, the blocks grew from 512 bytes to $4KB$ and more.

Advantages and disadvantages of increasing the block size:

- **Advantage:** Allows more information to be stored continuously.
  Explanation: The smaller the blocks, the greater the chance of breaking the storage sequence and splitting the data into blocks.

- **Disadvantage:** It's possible to store files smaller than the block size, thereby creating unused storage space.
  Explanation: A block is designed to hold data from only one file. Holding a quarter of a file in a block will cause waste (because $\frac{3}{4}$ of a block is lost).

Question: Why is block size important?
Answer: It enables better performance because there are fewer searches and more continuous information.[248] In case the file is large, it allows local sequential reading. In a small case, it is a waste.

Fragments

Dividing a block into fragments that allow storing the end of files. When a file occupying a fragment grows, the new information goes into the fragment in the block or is copied to a new block. Designed to save waste of storage, improve general performance and improve $lseek(2)$ system call performance.

Inodes

Inode is a reference, a pointer to a file. Each file has an inode. Contains information about the file that the operating system stores, excluding the file name. Each inode has a unique identification.

What information does inode contain?

- Permissions, user identification, group identification, etc.
- Timers – When the file was changed, when the permissions were changed, when it was last accessed, etc.
- Everything the $fstat/stat$ can receive - Information about the file, like for example, the size of the file.
- Direct pointer to a block.
- Indirect pointer to blocks.

Super-blocks

The super-block is a pointer to the first partition on the disk. Contains general information about the system, which includes the number of avaliable block and Inodes, timers, etc.

[248] For SSDs, there is no performance impact.

UFS structure

Maximal file size is defined as the maximum number of blocks that can be pointed to. Usually isn't limited to most UNIX systems, but in the past, there was a limitation between $2GB$ and a few TBs.

Directory is a file that contains a list of Inodes and their names respectively, such that we could access them. Can contains sub-directories. Build in the shape of a tree. Its content consists of $struct\ dirent$.[249]

Permissions for directories:

- **Read Access** – We can read using the $ls(1)$ cmd tool command.
- **Write Access** – We can create new files in the directory using the $touch(1)$ cmd tool command.
- **Execute Access** – We can do the $cd(1)$ cmd tool command.[250]

Hard Links – Two directories that point to the same Inode.[251]

Question: When is it useful?
Answer: When 2 users want to work on the same file each from their own folder. Also, when we have a binary file (like gcc) and depending on how it is called the compiler decides if it is gcc or $g++$.

Problem #1: When we use cloud storage and are using hard links that are shared between A and B. Who does pay the extra storage, A or B? How can we make sure not to double charge both of them on the same file.

Problem #2: Backup and restore can be misleading.[252]

Solution: A few years later, the soft links were established.

A **soft link** (or **symbolic link**) is a file that contains the path of another file.[253]

How did it actually solve the problems?

- **Solution for problem #1:** In the cloud, whoever owns the file will pay for the file, and the rest for the soft link only.
- **Solution for problem #2:** Backup and restore is not a problem as we restore the path to the file.

Question: What happens when the file in the path is deleted?
Answer: The soft link will become a broken link.

---

[249] $sturct\ dirent$ is a pointer to Inodes of a file.
[250] Change the working directory to this directory.
[251] When we delete the file from one place (deleting the hard link itself), its still exist, and decreases the pointer count by 1. Only when the last pointer is deleted will the block of the file be deleted.
[252] Did we actually create a new file?
[253] In Windows it's known as a shortcut.

Improvements that were proposed by Berkeley:

Berkeley offered few improvements to UFS.

- **They increased the block size** – Allowing for more contiguous data and better performance.
  Problem: Lack of disk utilization.
  Solution: They configured the fragments, which are sized between 512 bytes to $4KB$ and are intended to allow creating partitioned use of blocks. This makes the system 10 times faster, as we don't need to perform the seek operations too many times.

- **Duplicate super-block multiple times** – Allows a faster seek operation by accessing the closest super-block.

Catalog based File Systems:
Most of the modern file systems are catalog based. This catalog describes which files are on the system.[254] It's written consistently and consistently. Therefore, it is not suitable for an SSD that has wear.

Issues with the UFS model

There are a few issues with the UFS model:

- **No Log** – No documentation in case of a crash and malfunction so it is difficult to recover.[255]
- **Wear** – There are persistent writes to the catalog compared to other places on a file system, especially when the catalog is duplicated several times on the file system.
- **File Scattering** – When a file spreads and takes up a lot of blocks. Requires a seek operation per block.

[254] For example, in UNIX it's the super-block.
[255] For example, updating blocks of a file without updating the inode itself, which will cause the inode to point to other location. To solve this problem, we can create a file system log which records the actions we take and thus allows us to redo them or cancel them in the event of a malfunction.

EXT2 file system

$ext2$ is a file system that was written to the Linux kernel. It focuses on optimization.

What does it offer?

- Doubling the block size.
- There is no need for a super-block at the beginning and it is enough for each group of blocks to set a super-block for them only.
- To minimize fragmentation, in each write, 8 additional blocks are allocated.[256]

This leads to 70-80% of the optimal performance of the disk.

Partial struct of inode in EXT2:

```
struct ext2_inode {
        __le16 i_mode;          /* File mode */
        __le16 i_uid;           /* Low 16 bits of Owner Uid */
        __le32 i_size;          /* Size in bytes */
        __le32 i_atime;         /* Access time */
        __le32 i_ctime;         /* Creation time */
        __le32 i_mtime;         /* Modification time */
        __le32 i_dtime;         /* Deletion Time */
        __le16 i_gid;           /* Low 16 bits of Group Id */
        __le16 i_links_count;   /* Links count */
        __le32 i_blocks;        /* Blocks count */
        __le32 i_flags;         /* File flags */
..
}
```

Other file systems

- **Log Based File System** – A file system where we write every time to a log. There is no need for seek operations while writing, as the whole writing is done to a circular log.
- **Encrypted File System** – An encrypted file system that uses encryption[257] to encrypt the data of the files. Used to prevent Evil Maid Attack (EMA),[258] by encrypting the storage system, we prevent the computer's configuration from being changed.
  Question: Where does the encryption key come from?
  Answer: From the user, Trusted Platform Module (TPM)[259] or hardware security device.
- **Clustered File System** – Allows files to be saved on multiple computers with no size limit. Mainly intended for those dealing with Big Data.
- **ext3** – A file system that replaced $ext2$. Added a feature of a log. Was the default file system in old Linux versions.
- **ext4** – The current default file system in Linux machines.[260]
- **Virtual File System** – The way we refer to some file systems.[261]

---

[256] Reserved for the file.
[257] For example, AES.
[258] It's called that because if a maid comes to clean the room and you go down to the lobby, the maid can enter the BIOS and install all kinds of things on the computer.
[259] An international standard for a secure crypto processor, a dedicated microcontroller designed to secure hardware through integrated cryptographic keys.
[260] The features that this file system added are out of scope for this course.
[261] For example, to give the same machine drive C:\ and drive D:\, etc.

Assignments[262]

This section contains all the assignments that were given through the course, and the solutions for those assignments.

Solution for assignments 1 to 5 and to bonus assignments were written by Roy Simanovich and Linor Ronen. Solution for assignment 6 was written by Daniel Zinn. All the assignments were written in C programming language.

For your convenience, the assignments themselves are also provided. They were written by Dr. Nezer J. Zaidenberg and Mr. Arkady Gorodishtser from the department of Computer Science at Ariel University of Samaria.

The full source codes for the solutions of the assignments are available at GitHub:

- **Assignment 2 solution** – by Roy Simanovich and Linor Ronen
- **Assignment 3 solution** – by Roy Simanovich and Linor Ronen
- **Assignment 4 solution** – by Roy Simanovich and Linor Ronen
- **Assignment 5 solution** – by Roy Simanovich and Linor Ronen
- **Assignment 6 solution** – by Daniel Zinn
- **Assignment 4 bonus solution** – by Roy Simanovich and Linor Ronen
- **Assignment 5 bonus solution** – by Roy Simanovich and Linor Ronen

Solutions of assignments 2–5, 4 bonus and 5 bonus are licensed under **GNU General Public License version 3.0**. Full copy of the license is available in the links. Solution of assignment 6 is licensed under **GNU General Public License version 2.0**. Full copy of the license is available in the **following link**.

[262] Please note that the material for the bonus assignments isn't necessary for the exam, it's for enrichment only, and only brings bonus points to the final grade.

Assignment 1

Context

Congratulation! You are about to finished your Computer Science degree and looking for a job. A friend, working in a secret agency, suggested you submit your request, as they are looking for a new talent. In order to apply, you need to generate a secret key, based on your personal ID, private and family name. The organization will check if you are suitable for their need and invite you for an interview. The key generator is provided (named "$secret$") but hardcoded with company best talents names. Try to add yourself to the list and get a key for you. A test tool is also supplied (named "$decrept$"), don't hesitate to use it before submitting your request, as only one submission is allowed.

$secret$ executable partial source code (provided via $keyGenerator.c$):

```
#include "ecnript.h"
#include <stdio.h>

int main()
{

        int c = 0;

        while (c < 49 || c > 52){
                printf( "\n 1 - Michal Gor The Musician: ");
                printf( "\n 2 - Hilel Zak The Tolmid Hoham: ");
                printf( "\n 3 - Merav Verd The Engeneer: ");
                printf( "\n 4 - Tehila Kai The Creator: ");

                printf( "\n Select an agent :");
                c = getchar( );
        }


        char *name;
        long id;

        if (c == 49){
                name = "Michal Gor";
                id = 22446688;
        }

        if (c == 51){
                name = "Merav Verd";
                id = 11223344;
        }

        if (c == 50){
                name = "Hilel Zak";
                id = 11335577;
        }

        if (c == 52){
                name = "Tehila Kai";
                id = 321654987;
        }

        int type = getType(name);

        if (type != 1)
        {
                *((char*)-1) = 'x';
        }
        printf("starting encription\n");
        char* ecripted = getEncriptedMessage(name,type,id);
        printf("\nencription done:\n");
        printf("%s\n",ecripted);
        printf("****----------------------------------------****\n");
        printf("| Put the above in read.me, ZIP with your ID, and upload to moodle |\n");
        printf("****----------------------------------------****\n");
        return 0;
}
```

Solution

We'll use the GNU Debugger (GDB for short). First, we must load the executable to the debugger. We'll use the following command:

$$gdb\ ./secret$$

Then we'll set a breakpoint to line 49, which is the line right before the encryption starts:

$$break\ 49$$

After that, we'll execute the $secret$ program using the following command:

$$run$$

The program will stop at the breakpoint. At this point, all the necessary variables are already declared and aren't going to change for the rest of the program. Now we'll manipulate those variables:

$$set\ var\ name = \text{Your Name}$$
$$set\ var\ id = 123456789$$
$$set\ var\ type = getType(name)$$[263]

After that, we can continue to run the program using the following command:

$$continue$$

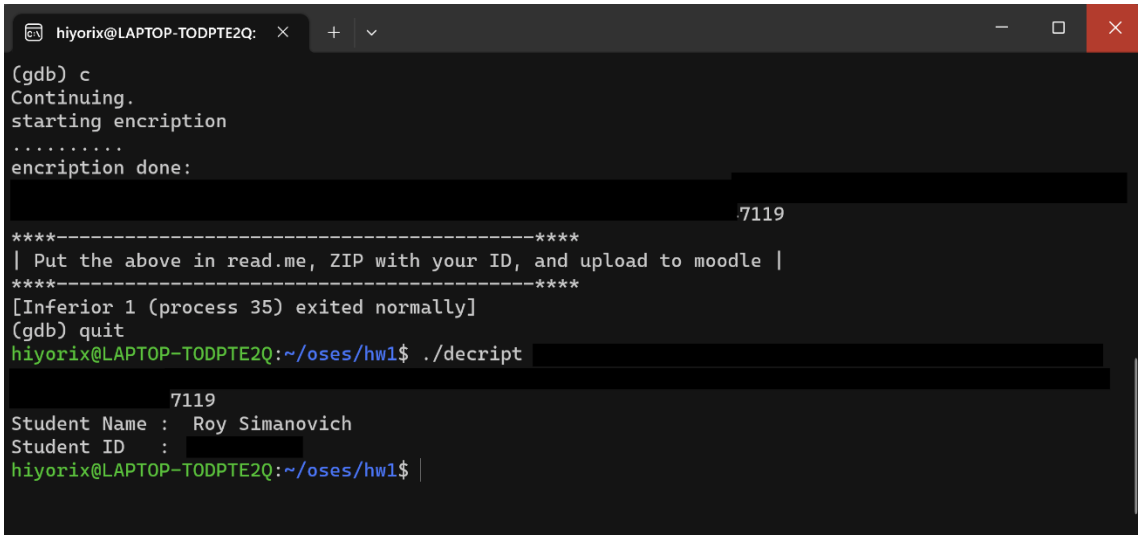
The secret program will generate an encrypted message, which we can decrypt using the $decrept$ program that's provided to use.

Now we can exit the debugger using the following command:

$$exit$$

If you'll follow the instructions, the $decrept$ program should print the values we entered in the first place.

Screenshot:



[263] The $getType()$ function is an internal function that used by the encryption library. It was used to make the decryption harder, and to make the program crash for harder reverse engineering.

Assignment 2

Context

The task is made of 3 parts: **files operations**,[264] **dynamic libraries**, and **basic shell**.

Part A (24 pts):

You are requested to implement two small programs that act like regular CMD tools.

- $\boldsymbol{cmp}$ – The tool will compare two files, and return "0" if they are equal, and "1" if not (return an INT). The tool will support $-v$ flag for verbose output. By this we mean that the tool will print "$equal$" or "$distinct$", in addition to returning the int value. The tool will support $-i$ flag, that mean "ignore case" so "AAA" and "aaa" meaning equals.
  Usage example: $cmp\ \ < file1 >< file2 >\ \ -v$
  Output: $equal$

- $\boldsymbol{copy}$ – The tool will copy a file to another place and/or name. The tool will return "0" on success, or "1" on failure (return an INT). The tool will create a new file, if it does not exist, but it will not overwrite a file if it does exist. The tool will support $-v$ flag, that will output "$success$" if the file is copied, or "$target\ file\ exist$" if this is the case, or "$general\ failure$" on some other problem.[265] The tool will support $-f$ flag (that means force), that allows to overwrite the target file.
  Usage example: $copy\ \ < file1 >< file2 >\ \ -v$
  Output: $success$

Part B (24 pts):

You are requested to implement a coding library. We have two codding methods:

- **Method A, named $\boldsymbol{codecA}$** – Covert all lowercase characters to upper case, and all upper case to lower case. All other characters will remain unchanged.
- **Method B, named $\boldsymbol{codecB}$** – Convert all characters to the 3-rd next character.[266]

The libraries should support "encode" and "decode" methods.[267]

1. Write 2 different shared libraries, each implementing its algorithm.
2. Write two tools, named encode and decode, to utilize the libraries. The tools will get some text and convert it according to selected library.

Usage: $encode/decode\ \ < codec >< message >$

Output: Encoded/decoded string.

Example: "$encode\ codecA\ aaaBBB$" will return "$AAAbbb$".

Example: "$decode\ codecB\ EEEddd$" will return "$BBBaaa$".

---

[264] If the must-have parameters are missing (the file names) return 1 and print a usage explanation and the optional flags.
[265] In addition to the returned INT value.
[266] Adding a number of 3 to the ASCII value.
[267] The libraries should be "reversable", meaning that if one does "encode" and then "decode", he will get the original string.

Part C (52 pts):

You are requested to write a shell program named $stshell$ (st for students).

The features of the shell are:

- Be able to run CMD tools that exist on system.[268]
- Be able to stop running tool by pressing $Ctrl + C$, but not killing the shell itself.[269]
- Be able to redirect output with "$>$" and "$>>$", and allow piping with "|", at least for 2 following pipes.[270]
- Be able to stop itself by "$exit$" command.

[268] By using $fork(2)$, $exec(2)$ and $wait(2)$ system calls.
[269] By signal handler.
[270] For example, command like this should be supported "$ls\ -l \mid grep\ aaa \mid wc$".

Solution

***cmp.c*:**

Enumeration explanation:

- ***OPT_NONE*** – No options.[271]
- ***OPT_IGNORE_CASE*** – Ignore case when comparing.
- ***OPT_VERBOSE*** – Verbose output.[272]
- ***EQUAL*** – The files are equal.
- ***DISTINCT*** – The files are distinct, or an error occurred.[273]

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>
typedef enum {
    OPT_NONE = 0,
    OPT_IGNORE_CASE = 1,
    OPT_VERBOSE = 2
} cmp_options;
typedef enum {
    EQUAL = 0,
    DISTINCT = 1
} cmp_result;
int main(int argc, char** argv) {
    FILE *f1 = NULL, *f2 = NULL;
    cmp_options options = OPT_NONE;
    char c1 = '\0', c2 = '\0';
    bool equal = true;
    if (argc < 3 || argc > 5) {
        fprintf(stderr, "Usage: %s file1 file2 [-iv]\n", *(argv));
        return (int)DISTINCT;
    }
    for (int i = 3; i < argc; ++i) {
        if (strcmp(*(argv + i), "-i") == 0)
            options |= OPT_IGNORE_CASE;
        else if (strcmp(*(argv + i), "-v") == 0)
            options |= OPT_VERBOSE;
        else {
            fprintf(stderr, "[CMP] Unknown option \"%s\".\n", *(argv + i));
            return (int)DISTINCT;
        }
    }
    f1 = fopen(*(argv + 1), "rb");
    f2 = fopen(*(argv + 2), "rb");
    if (f1 == NULL) {
        if (options & OPT_VERBOSE)
            fprintf(stderr, "[CMP] Error opening file \"%s\" for reading.\n", *(argv + 1));
        return (int)DISTINCT;
    }
    if (f2 == NULL) {
        if (options & OPT_VERBOSE)
            fprintf(stderr, "[CMP] Error opening file \"%s\" for reading.\n", *(argv + 2));
        return (int)DISTINCT;
    }
    fseek(f1, 0, SEEK_END);
    fseek(f2, 0, SEEK_END);
    if (ftell(f1) != ftell(f2)) {
        if (options & OPT_VERBOSE)
            fprintf(stdout, "distinct\n");
        fclose(f1);
        fclose(f2);
        return (int)DISTINCT;
    }
    fseek(f1, 0, SEEK_SET);
    fseek(f2, 0, SEEK_SET);
    while (equal) {
        c1 = fgetc(f1);
        c2 = fgetc(f2);
        if (c1 == EOF || c2 == EOF)
            break;
        if (options & OPT_IGNORE_CASE) {
            c1 = tolower(c1);
            c2 = tolower(c2);
        }
        if (c1 != c2)
            equal = false;
    }
    fclose(f1);
    fclose(f2);
    if (!equal) {
        if (options & OPT_VERBOSE)
            fprintf(stdout, "distinct\n");
        return (int)DISTINCT;
    }
    if (options & OPT_VERBOSE)
        fprintf(stdout, "equal\n");
    return (int)EQUAL;
}
```

***copy.c*:**

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>
typedef enum {
    OPT_NONE = 0,
    OPT_VERBOSE = 1,
    OPT_FORCE = 2
} copy_options;
typedef enum {
    SUCCESS = 0,
    FAILURE = 1
} copy_result;
int main(int argc, char** argv) {
    FILE *f1 = NULL, *f2 = NULL;
    copy_options options = OPT_NONE;
    char c = '\0';
    if (argc < 3 || argc > 5) {
        fprintf(stderr, "Usage: %s <source> <destination> [-vf]\n", *(argv));
        return (int)FAILURE;
    }
    for (int i = 3; i < argc; ++i) {
        if (strcmp(*(argv + i), "-v") == 0)
            options |= OPT_VERBOSE;
        else if (strcmp(*(argv + i), "-f") == 0)
            options |= OPT_FORCE;
        else {
            if (options & OPT_VERBOSE)
                fprintf(stderr, "general failure\n");
            return (int)FAILURE;
        }
    }
    f1 = fopen(*(argv + 1), "r");
    if (f1 == NULL) {
        if (options & OPT_VERBOSE)
            fprintf(stderr, "general failure\n");
        return (int)FAILURE;
    }
    if (!(options & OPT_FORCE)) {
        f2 = fopen(*(argv + 2), "rb");
        if (f2 != NULL) {
            if (options & OPT_VERBOSE)
                fprintf(stderr, "target file exist\n");
            fclose(f1);
            fclose(f2);
            return (int)FAILURE;
        }
    }
    f2 = fopen(*(argv + 2), "wb");
    if (f2 == NULL) {
        if (options & OPT_VERBOSE)
            fprintf(stderr, "general failure\n");
        return (int)FAILURE;
    }
    while ((c = fgetc(f1)) != EOF)
        fputc(c, f2);
    fclose(f1);
    fclose(f2);
    if (options & OPT_VERBOSE)
        fprintf(stdout, "success\n");
    return (int)SUCCESS;
}
```

[271] Default option.
[272] Print equal or distinct.
[273] e.g., file not found.

Enumeration explanation:

- $\boldsymbol{OPT\_NONE}$ – No options.[274]
- $\boldsymbol{OPT\_VERBOSE}$ – Verbose output.[275]
- $\boldsymbol{OPT\_FORCE}$ – Force the copy even if the file already exists.
- $\boldsymbol{SUCCESS}$ – The copy was successful.
- $\boldsymbol{FAILURE}$ – An error occurred.[276]

***codec.h*:**

```
#define U_CHAR_CASE 0x20
#define U_CHAR_ADDITION 0x03
#define U_CHAR_MODULUS 0xFF

unsigned char* encode(unsigned char* input, size_t len);
unsigned char* decode(unsigned char* input, size_t len);
```

Explanation:

- U_CHAR_CASE – The case value is used to determine if a character is upper or is in lower case.
- $U\_CHAR\_ADDITION$ – The addition/subtraction value is used to determine if a character is upper or lower case.
- $U\_CHAR\_MODULUS$ – The modulus value for the encoding/decoding. Used to avoid overflow.

The $encode(unsigned\ char *\ input, size\_t\ len)$ function is a template for encoding a string. $input$ is the string to encode. len is length of the string. The function returns the encoded string or NULL if malloc failed.

The $decode(unsigned\ char *\ input, size\_t\ len)$ function is a template for decoding a string. $input$ is the string to decode. len is length of the string. The function returns the decoded string or NULL if malloc failed.

***codecA.c*:**

```
#include <stdio.h>
#include <stdlib.h>
#include "codec.h"
unsigned char* encode(unsigned char* input, size_t len) {
    unsigned char* output = (unsigned char*) malloc(len + 1);
    if (output == NULL)
        return NULL;
    for (size_t i = 0; i < len; ++i) {
        unsigned char tmp = *(input + i);
        if (tmp > 0x40 && tmp < 0x5B)
            tmp += U_CHAR_CASE;
        else if (tmp > 0x60 && tmp < 0x7B)
            tmp -= U_CHAR_CASE;
        *(output + i) = tmp;
    }
    *(output + len) = '\0';
    return output;
}
unsigned char* decode(unsigned char* input, size_t len) {
    return encode(input, len);
}
```

[274] Default option.
[275] Print equal or distinct.
[276] e.g., file not found.

codecB.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "codec.h"
unsigned char* encode(unsigned char* input, size_t len) {
   unsigned char* output = (unsigned char*) malloc(len + 1);
   if (output == NULL)
      return NULL;
   for (size_t i = 0; i < len; ++i)
      *(output + i) = ((*(input + i) + U_CHAR_ADDITION) % U_CHAR_MODULUS);
   *(output + len) = '\0';
   return output;
}
unsigned char* decode(unsigned char* input, size_t len) {
   unsigned char* output = (unsigned char*) malloc(len + 1);
   if (output == NULL)
      return NULL;
   for (size_t i = 0; i < len; ++i)
      *(output + i) = ((*(input + i) - U_CHAR_ADDITION) % U_CHAR_MODULUS);
   *(output + len) = '\0';
   return output;
}
```

encode.c:

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <dlfcn.h> // for dlopen, dlsym, dlclose
typedef enum {
   SUCCESS = 0,
   FAILURE = 1
} encode_result;
unsigned char* (*encode)(unsigned char*, size_t);
bool isCodec(char* codec) {
   return strcmp(codec, "codecA") == 0 || strcmp(codec, "codecB") == 0;
}
int main(int argc, char** argv) {
   void* handle = NULL;
   unsigned char* encoded = NULL;
   size_t len = 0;
   if (argc != 3) {
      fprintf(stderr, "Usage: %s <codec type> <text>\n", *(argv));
      return (int)FAILURE;
   }
   if (!isCodec(*(argv + 1))) {
      fprintf(stderr, "[Encoder] Error: codec type must be either codecA or codecB (case senestive).\n");
      return (int)FAILURE;
   }
   if ((len = strlen(*(argv + 2))) == 0) {
      fprintf(stderr, "[Encoder] Error: text must not be empty.\n");
      return (int)FAILURE;
   }
   if ((handle = dlopen((strcmp(*(argv + 1), "codecA") == 0 ? "./libcodecA.so":"./libcodecB.so"),
RTLD_LAZY)) == NULL) {
      fprintf(stderr, "[Encoder] Error: %s\n", dlerror());
      return (int)FAILURE;
   }
   if ((encode = dlsym(handle, "encode")) == NULL) {
      fprintf(stderr, "[Encoder] Error: %s\n", dlerror());
      return (int)FAILURE;
   }
   if ((encoded = encode((unsigned char*)*(argv + 2), len)) == NULL) {
      fprintf(stderr, "[Encoder] Error: Falied to malloc.\n");
      return (int)FAILURE;
   }
   fprintf(stdout, "%s\n", encoded);
   dlclose(handle);
   free(encoded);
   return (int)SUCCESS;
}
```

Explanation:

- $\boldsymbol{SUCCESS}$ – The encode was successful.
- $\boldsymbol{FAILURE}$ – An error occurred.[277]

The $(*\,encode)(unsigned\ char\ *, size\_t)$ typedef is a macro that used for the dynamically loaded function from a shared library. This function is used to encode a given text using a given codec.

decode.c:

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <dlfcn.h> // for dlopen, dlsym, dlclose
typedef enum {
    SUCCESS = 0,
    FAILURE = 1
} decode_result;
unsigned char* (*decode)(unsigned char*, size_t);
bool isCodec(char* codec) {
    return strcmp(codec, "codecA") == 0 || strcmp(codec, "codecB") == 0;
}
int main(int argc, char** argv) {
    void* handle = NULL;
    unsigned char* decoded = NULL;
    size_t len = 0;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <codec type> <text>\n", *(argv));
        return (int)FAILURE;
    }
    if (!isCodec(*(argv + 1))) {
        fprintf(stderr, "[Decoder] Error: codec type must be either codecA or codecB (case senestive).\n");
        return (int)FAILURE;
    }
    if (!(len = strlen(*(argv + 2)))) {
        fprintf(stderr, "[Decoder] Error: text must not be empty.\n");
        return (int)FAILURE;
    }
    if ((handle = dlopen((strcmp(*(argv + 1), "codecA") == 0 ? "./libcodecA.so":"./libcodecB.so"), RTLD_LAZY)) == NULL) {
        fprintf(stderr, "[Decoder] Error: %s\n", dlerror());
        return (int)FAILURE;
    }
    if ((decode = dlsym(handle, "decode")) == NULL) {
        fprintf(stderr, "[Decoder] Error: %s\n", dlerror());
        return (int)FAILURE;
    }
    if ((decoded = decode((unsigned char*)*(argv + 2), len)) == NULL) {
        fprintf(stderr, "[Decoder] Error: Falied to malloc.\n");
        return (int)FAILURE;
    }
    fprintf(stdout, "%s\n", decoded);
    dlclose(handle);
    free(decoded);
    return (int)SUCCESS;
}
```

[277] e.g., library not found, wrong arguments.

Explanation:

- $\boldsymbol{SUCCESS}$ – The decode was successful.
- $\boldsymbol{FAILURE}$ – An error occurred.[278]

The $(*\ decode)(unsigned\ char\ *, size\_t)$ typedef is a macro that used for the dynamically loaded function from a shared library. This function is used to decode a given text using a given codec.

$\boldsymbol{stshell.h}$:

```
typedef enum _CommandType
{
   Internal = 0,
   External = 1
} CommandType;
typedef enum _Result
{
   Success = 0,
   Failure = 1
} Result;
char *append(char before, char *str, char after);
CommandType parse_command(char* command, char** argv);
Result cmdCD(char *path, int argc);
void execute_command(char** argv);
```

Enumeration explanation:

- $\boldsymbol{Internal}$ – Indicates a shell internal command.
- $\boldsymbol{External}$ – Indicates a shell external command (using execvp(2) system call).
- $\boldsymbol{Success}$ – Operation was successful.
- $\boldsymbol{Failure}$ – An error occurred.

Functions explanation:

- The $append(char\ before, char\ *str, char\ after)$ function appends a character to a string. $before$ is the character to append before the string. str is the string to append to. $after$ is the character to append after the string. The function returns the new string.
- The $parse\_command(char\ *\ command, char\ **\ argv)$ function parses a command into arguments, by also supporting quotes and internal commands. $command$ is the command to parse. $argv$ is the array of arguments. The function returns 0 if it's an internal command, 1 if it's an external command. This function will modify the $argv$ array and execute internal commands.
- The $cmdCD(char\ *path, int\ argc)$ function executes change directory command. $path$ is the path to change to. $argc$ is the number of arguments. The function returns Success if the command succeeded, Failure otherwise. Number of arguments must be 1.
- The $execute\_command(char\ **\ argv)$ function executes a command. $argv$ is the array of arguments.

[278] e.g., library not found, wrong arguments.

Partial source code of $stshell.c$:[279]

Headers, global variables, $main()$, $append()$ and $cmdCD()$ functions:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <fcntl.h>
#include <errno.h>
#include <strings.h>
#include <string.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/utsname.h>
#include "stshell.h"
char *homedir = NULL;
char *workingdir = NULL;
char *cwd = NULL;
int main(void) {
	char **argv = NULL;
	char command[MAX_COMMAND_LENGTH + 1] = {0};
	struct utsname sysinfo;
	uname(&sysinfo);
	cwd = (char *)calloc((MAX_PATH_LENGTH + 1), sizeof(char));
	workingdir = (char *)calloc((MAX_PATH_LENGTH + 1), sizeof(char));
	argv = (char **)calloc((MAX_ARGS + 1), sizeof(char *));
	homedir = getenv("HOME");
	signal(SIGINT, SIG_IGN);
	while (1) {
		for (size_t i = 0; i < MAX_ARGS; ++i)
			*(argv + i) = (char *)calloc((MAX_ARG_LENGTH + 1), sizeof(char));
		char *user_name = getenv("USER");
		char *cwd_changed = (char *)calloc((MAX_PATH_LENGTH + 1), sizeof(char));
		strcpy(cwd_changed + 1, cwd + strlen(homedir));
		*cwd_changed = '~';
		fprintf(stdout, "%s@%s:%s%s", user_name, sysinfo.nodename, (strstr(cwd, homedir) == NULL ? cwd : cwd_changed), "$ ");
		fflush(stdout);
		free(cwd_changed);
		fgets(command, MAX_COMMAND_LENGTH, stdin);
		if (parse_command(command, argv) == Internal) {
			for (size_t i = 0; i < MAX_ARGS; ++i)
				free(*(argv + i));
			continue;
		}
		execute_command(argv);
		for (size_t i = 0; i < MAX_ARGS; ++i) {
			if (*(argv + i) != NULL)
				free(*(argv + i));
		}
		bzero(command, (MAX_COMMAND_LENGTH + 1));
	}
	free(argv);
	free(cwd);
	return EXIT_SUCCESS;
}
char *append(char before, char *str, char after) {
	size_t len = strlen(str);
	if (before) {
		for (size_t i = len; i > 0; --i)
			*(str + i) = *(str + i - 1);
		*str = before;
	}
	if (after) {
		*(str + len) = after;
		*(str + len + 1) = '\0';
	}
	return str;
}
Result cmdCD(char *path, int argc) {
	if (argc == 2) {
		if (strcmp(path, "~") == 0) {
			getcwd(workingdir, MAX_PATH_LENGTH);
			return Success;
		}

		else if (strcmp(path, "-") == 0) {
			chdir(workingdir);
			getcwd(workingdir, MAX_PATH_LENGTH);
			return Success;
		}
		getcwd(workingdir, MAX_PATH_LENGTH);
		chdir(path);
	} else {
		getcwd(workingdir, MAX_PATH_LENGTH);
		chdir(homedir);
	}
	return Success;
}
```

[279] This code is missing critical and vital error checks. Also its missing support for both piping and redirect in the same command.

$parse\_command()$ function:

```
CommandType parse_command(char *command, char **argv) {
        char **pargv = argv;
        char *token = NULL;
        int words = 1;
        bool inQuotes = false;
        command = strtok(command, "\n");
        if (command == NULL || strlen(command) == 0) return Internal;
        for (size_t k = 0; k < strlen(command); ++k) {
                if (*(command + k) == '"') inQuotes = !inQuotes;
                if (!inQuotes && *(command + k) == ' ' && *(command + k + 1) != ' ') ++words;
        }
        token = strtok(command, " ");
        if (strcmp(token, CMD_EXIT) == 0)
                exit(EXIT_SUCCESS);
        else if (strcmp(token, CMD_CD) == 0) {
                token = strtok(NULL, " ");
                cmdCD(token, words);
                return Internal;
        }
        else if (strcmp(token, CMD_CLEAR) == 0) {
                write(STDOUT_FILENO, CMD_CLEAR_FLUSH, CMD_CLEAR_FLUSH_LEN);
                return Internal;
        }
        else if (strcmp(token, CMD_PWD) == 0) {
                fprintf(stdout, "%s\n", cwd);
                return Internal;
        }
        while (token != NULL) {
                if (*token == '"' && strlen(token) > 1) {
                        char *tmp = (char *)calloc((MAX_ARG_LENGTH + 1), sizeof(char));
                        strcpy(tmp, (token + 1));
                        while (token != NULL) {
                                token = strtok(NULL, "\"");
                                if (token != NULL) {
                                        strcat(tmp, " ");
                                        strcat(tmp, token);
                                }
                        }
                        if (*(tmp + strlen(tmp) - 1) == '"')
                                *(tmp + strlen(tmp) - 1) = '\0';
                        memcpy(*pargv, tmp, strlen(tmp));
                        ++pargv;
                        token = strtok(NULL, " ");
                        free(tmp);
                        continue;
                }
                memcpy(*pargv, token, strlen(token));
                ++pargv;
                token = strtok(NULL, " ");
        }
        if (*pargv != NULL) {
                free(*pargv);
                *pargv = NULL;
        }
        if (strcmp(*argv, CMD_CMP) == 0 || strcmp(*argv, CMD_COPY) == 0 || strcmp(*argv, CMD_ENCODE) == 0 || strcmp(*argv, CMD_DECODE) == 0) {
                *argv = append('/', *argv, '\0');
                *argv = append('.', *argv, '\0');
                execute_command(argv);
                return Internal;
        }
        return External;
}
```

***execute_command*()** function:

```
void execute_command(char **argv) {
	pid_t pid;
	int status, i = 0, num_pipes = 0;
	bool redirect = false;
	while (*(argv + i) != NULL) {
		if (strcmp(*(argv + i), "|") == 0) num_pipes++;
		else if (strcmp(*(argv + i), ">") == 0 || strcmp(*(argv + i), ">>") == 0 || strcmp(*(argv + i), "<") == 0) redirect = true;
		i++;
	}
	if (num_pipes == 0 && !redirect) {
		pid = fork();
		if (pid == 0) {
			signal(SIGINT, SIG_DFL);
			execvp(*argv, argv);
		} else waitpid(pid, &status, 0);
	} else {
		pid = fork();
		if (pid == 0) {
			char **argv_cpy = (char **)calloc((MAX_ARGS + 1), sizeof(char *));
			int input_fd = 0, output_fd = 1, append_fd = 1; // Input and output file descriptors.
			memcpy(argv_cpy, argv, (MAX_ARGS + 1) * sizeof(char *));
			signal(SIGINT, SIG_DFL);
			if (num_pipes == 0 && redirect) {
				for (i = 0; (i < MAX_ARGS) && (*(argv + i) != NULL); ++i) {
					if (strcmp(*(argv + i), "<") == 0) {
						input_fd = open(*(argv + i + 1), O_RDONLY);
						dup2(input_fd, STDIN_FILENO);
						close(input_fd);
						*(argv_cpy + i) = NULL;
					} else if (strcmp(*(argv + i), ">") == 0){
						output_fd = open(*(argv + i + 1), O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
						dup2(output_fd, STDOUT_FILENO);
						close(output_fd);
						*(argv_cpy + i) = NULL;
					} else if (strcmp(*(argv + i), ">>") == 0) {
						append_fd = open(*(argv + i + 1), O_WRONLY | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR);
						dup2(append_fd, STDOUT_FILENO);
						close(append_fd);
						*(argv_cpy + i) = NULL;
					}
					i++;
				}
				execvp(*argv_cpy, argv_cpy);
			} else if (num_pipes == 1) {
				int pipe_fd[2]; // Pipe file descriptors.
				pipe(pipe_fd);
				for (i = 0; i < MAX_ARGS && (*(argv + i) != NULL); ++i) {
					if (strcmp(*(argv + i), "|") == 0) {
						*(argv_cpy + i) = NULL;
						break;
					}
				}
				pid = fork();
				if (pid == 0) {
					dup2(*(pipe_fd + 1), STDOUT_FILENO);
					close(*pipe_fd);
					close(*(pipe_fd + 1));
					execvp(*argv_cpy, argv_cpy);
				} else {
					dup2(*pipe_fd, STDIN_FILENO);
					close(*pipe_fd);
					close(*(pipe_fd + 1));
					execvp(*(argv_cpy + i + 1), (argv_cpy + i + 1));
				}
			} else if (num_pipes == 2) {
				int pipe_fd[4]; // Pipe file descriptors.
				pipe(pipe_fd);
				pipe(pipe_fd + 2);
				int loc_pipe1 = 0, loc_pipe2 = 0; // Location of the first and second pipe.
				for (i = 0; i < MAX_ARGS && (*(argv + i) != NULL); ++i) {
					if (strcmp(*(argv + i), "|") == 0) {
						*(argv_cpy + i) = NULL;
						if (loc_pipe1 == 0) loc_pipe1 = i;
						else {
							loc_pipe2 = i;
							break;
						}
					}
				}
				pid = fork();
				if (pid == 0) {
					dup2(*(pipe_fd + 1), STDOUT_FILENO);
					close(*pipe_fd);
					close(*(pipe_fd + 1));
					close(*(pipe_fd + 2));
					close(*(pipe_fd + 3));
					execvp(*argv_cpy, argv_cpy);
				}
				pid = fork();
				if (pid == 0) {
					dup2(*pipe_fd, STDIN_FILENO);
					dup2(*(pipe_fd + 3), STDOUT_FILENO);
					close(*pipe_fd);
					close(*(pipe_fd + 1));
					close(*(pipe_fd + 2));
					close(*(pipe_fd + 3));
					execvp(*(argv_cpy + loc_pipe1 + 1), (argv_cpy + loc_pipe1 + 1));
				} else {
					dup2(*(pipe_fd + 2), STDIN_FILENO);
					close(*pipe_fd);
					close(*(pipe_fd + 1));
					close(*(pipe_fd + 2));
					close(*(pipe_fd + 3));
					execvp(*(argv_cpy + loc_pipe2 + 1), (argv_cpy + loc_pipe2 + 1));
				}
			}
		} else waitpid(pid, &status, 0);
	}
}
```

Assignment 3

Context

This time, we are dealing with communications. Your goal is to implement a chat tool that will help to set some performance tests.

Part A – Basic Functionality (25 pts)

Implement a chat cmd tool that can send messages over the network, to the same tool, listening on the other side, and get the response, so there will be 2 sides communication, simultaneously. Name the tool $stnc$.[280]

Usage:

- **The client side:** $./stnc\ -c\ IP\ PORT$
- **The server side:** $./stnc\ -s\ PORT$

The communication will be done using IPv4 TCP protocol. By "chat" we mean a tool that can read input from keyboard, and at the same time listen to the socket of the other side.

Part B – Performance test (75 pts)

Extend your tool, to make it a network performance test utility. You can use your chat channel for system communication, like transferring states, times, etc.

By performance test we mean the next tasks:

1. Generate a chunk of data, with 100MB size.
2. Generate a checksum (hash) for the data above.
3. Transmit the data with selected communication style, while measuring the time it takes.
4. Report the result to $stdout$.

Communications styles are:

- TCP/UDP in IPv4/IPv6 (4 variants)
- mmap a file, named pipe (2 variants)
- Unix Domain Socket (UDS): stream and datagram (2 variants)

Usage:

- **The client side:** $./stnc\ -c\ IP\ PORT\ -p\ <type><param>$
  - $-p$ will indicate to perform the test
  - $<type>$ will be the communication types. It can be $ipv4$, $ipv6$, $mmap$, $pipe$ or uds.
  - $<param>$ will be a parameter for the type. It can be udp/tcp or $dgram/stream$ or a file name.[281]
- The server side: $./stnc\ -s\ PORT\ -p\ -q$
  - $-p$ flag will let you know that we are going to test the performance.
  - $-q$ flag will enable quiet mode, in which only testing results will be printed.[282]

---

[280] st for students, nc for network communication.

[281] You have only 8 combinations: $ipv4\ tcp$, $ipv4\ udp$, $ipv6\ tcp$, $ipv6\ udp$, $uds\ dgram$, $uds\ stream$, $mmap\ filename$, and $pipe\ filename$.

[282] This is extremely important for the automatic test.

The results will be in milliseconds (ms) and printed like this:

$$name\_type, time$$

Examples:

- $ipv4\_udp, 112233$
- $uds\_stream, 112233$
- $mmap, 223355$
- $pipe, 554411$

Notes:

- It's clear that the performance test tool is NOT a chat tool, so there is no expectation to have a conversation before or after the test. Both the client and the server will finish once the test of the requested type is done.
- If there is an error in parameters, like missing params, or giving some malformed text, print the help/usage text (in any form) and exit.
- It's highly suggested to use the Beej IPC Guide.
- The task is meant to be done with $poll(2)$ or $select(2)$ APIs, and without threads.

Solution[283]

***stnc.c*:**

```
#include "stnc.h"
#include <stdlib.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
int main(int argc, char **argv) {
	if (argc < 2)
	{
		stnc_print_usage(*argv, 0);
		return EXIT_FAILURE;
	} else if (strcmp(*(argv + 1), "-c") == 0) {
		if (argc < 4 || argc > 7) {
			stnc_print_usage(*argv, 1);
			return EXIT_FAILURE;
		} else if (argc == 4) {
			stnc_print_license();
			fprintf(stdout, "Client chat mode\n");
			return stnc_client_chat(*(argv + 2), *(argv + 3));
		} else if (strcmp(*(argv + 4), "-p") == 0) {
			if (argc != 7) {
				stnc_print_usage(*argv, 1);
				return EXIT_FAILURE;
			}
			stnc_print_license();
			fprintf(stdout, "Client performance mode\n");
			return stnc_client_performance(*(argv + 2), *(argv + 3), *(argv + 5), *(argv + 6), CLIENT_QUIET_MODE);
		} else {
			stnc_print_usage(*argv, 1);
			return EXIT_FAILURE;
		}
	} else if (strcmp(*(argv + 1), "-s") == 0) {
		if (argc < 3 || argc > 5) {
			stnc_print_usage(*argv, 2);
			return EXIT_FAILURE;
		} else if (argc == 3) {
			stnc_print_license();
			fprintf(stdout, "Server chat mode\n");
			return stnc_server_chat(*(argv + 2));
		} else if (strcmp(*(argv + 3), "-p") == 0) {
			bool quietmode = false;
			if (argc != 4 && argc != 5) {
				stnc_print_usage(*argv, 2);
				return EXIT_FAILURE;
			}
			else if (argc == 5) {
				if (strcmp(*(argv + 4), "-q") == 0)
					quietmode = true;
				else {
					stnc_print_usage(*argv, 2);
					return EXIT_FAILURE;
				}
			}
			if (!quietmode) {
				stnc_print_license();
				fprintf(stdout, "Server performance mode\n");
			}
			return stnc_server_performance(*(argv + 2), quietmode);
		}
		else {
			stnc_print_usage(*argv, 2);
			return EXIT_FAILURE;
		}
	} else {
		stnc_print_usage(*argv, 0);
		return EXIT_FAILURE;
	}
	return EXIT_SUCCESS;
}
```

[283] The codes in the solution don't have error checking for simplicity. Full code is available in GitHub.

stnc_chat.c:

```
#include "stnc.h"
#include <stdlib.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <poll.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
int stnc_client_chat(char *ip, char *port) {
	struct sockaddr_in server;
	char buffer[MAX_MESSAGE_SIZE] = {0};
	uint16_t portNumber = atoi(port);
	ssize_t writeBytes = 0, readBytes = 0;
	int sockfd = socket(AF_INET, SOCK_STREAM, 0);
	memset(&server, 0, sizeof(server));
	server.sin_family = AF_INET;
	server.sin_port = htons(portNumber);
	inet_pton(AF_INET, ip, &server.sin_addr);
	connect(sockfd, (struct sockaddr *)&server, sizeof(server));
	fprintf(stdout, "Connection established to %s:%s\n", ip, port);
	struct pollfd pfds[2];
	pfds[0].fd = STDIN_FILENO;
	pfds[0].events = POLLIN;
	pfds[1].fd = sockfd;
	pfds[1].events = POLLIN;
	while (true) {
		poll(pfds, 2, -1);
		if (pfds[0].revents & POLLIN) {
			fgets(buffer, MAX_MESSAGE_SIZE, stdin);
			buffer[strlen(buffer) - 1] = '\0';
			writeBytes = send(sockfd, buffer, strlen(buffer), 0);
			if (writeBytes <= 0) {
				fprintf(stdout, "Connection closed by the peer.\n");
				break;
			}
			memset(buffer, 0, MAX_MESSAGE_SIZE);
		}
		if (pfds[1].revents & POLLIN) {
			readBytes = recv(sockfd, buffer, MAX_MESSAGE_SIZE, 0);
			if (readBytes <= 0) {
				fprintf(stdout, "Connection closed by the peer.\n");
				break;
			}
			fprintf(stdout, "Peer: %s\n", buffer);
			memset(buffer, 0, MAX_MESSAGE_SIZE);
		}
	}
	close(sockfd);
	return EXIT_SUCCESS;
}
int stnc_server_chat(char *port) {
	struct sockaddr_in server, client;
	char buffer[MAX_MESSAGE_SIZE] = {0};
	uint16_t portNumber = atoi(port);
	socklen_t clientLen = sizeof(client);
	ssize_t writeBytes = 0, readBytes = 0;
	int sockfd = INVALID_SOCKET, clientfd = INVALID_SOCKET, reuse = 1;
	sockfd = socket(AF_INET, SOCK_STREAM, 0);
	setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
	memset(&server, 0, sizeof(server));
	memset(&client, 0, sizeof(client));
	server.sin_family = AF_INET;
	server.sin_addr.s_addr = htonl(INADDR_ANY);
	server.sin_port = htons(portNumber);
	bind(sockfd, (struct sockaddr *)&server, sizeof(server));
	listen(sockfd, 1);
	fprintf(stdout, "Waiting for incoming connection...\n");
	while (true) {
		clientfd = accept(sockfd, (struct sockaddr *)&client, (socklen_t *)&clientLen);
		fprintf(stdout, "Connection established with %s:%d\n", inet_ntoa(client.sin_addr), ntohs(client.sin_port));
		struct pollfd pfds[2];
		pfds[0].fd = STDIN_FILENO;
		pfds[0].events = POLLIN;
		pfds[1].fd = clientfd;
		pfds[1].events = POLLIN;
		while (true) {
			poll(pfds, 2, -1);
			memset(buffer, 0, MAX_MESSAGE_SIZE);
			if (pfds[1].revents & POLLIN) {
				readBytes = recv(clientfd, buffer, MAX_MESSAGE_SIZE, 0);
				if (readBytes <= 0) {
					fprintf(stdout, "Connection closed by the peer.\n");
					break;
				}
				fprintf(stdout, "Peer: %s\n", buffer);
			}
			if (pfds[0].revents & POLLIN) {
				fgets(buffer, MAX_MESSAGE_SIZE, stdin);
				buffer[strlen(buffer) - 1] = '\0';
				writeBytes = send(clientfd, buffer, strlen(buffer), 0);
				if (writeBytes == 0) {
					fprintf(stdout, "Connection closed by the peer.\n");
					break;
				}
			}
		}
		close(clientfd);
	}
	close(sockfd);
	return EXIT_SUCCESS;
}
```

Performance Mode solution:

The STNC protocol is a custom protocol we built for part B of the assignment, which is an application header, and it supports the STNC performance test by providing an efficient way to transmit important data.

The STNC packet header, which is sized only 8 bytes, follow this structure:

Bits 0 to 7	Bits 8 to 15	Bits 16 to 23	Bits 24 to 31
Message Type	Protocol	Param	Error Code
Payload Size			
Payload			

<u>Breakdown of each parameter</u>:

- **Message Type** (1 byte): This field indicates the STNC packet type. Options: $MSGT\_INIT$, $MSGT\_ACK$ and $MSGT\_DATA$. This field is mandatory for all messages.
    - $MSGT\_INIT$ (Initialization) – Communication started, exchange transfer type. This message is sent only once, at the beginning of the communication. Sent by the client only.
    - $MSGT\_ACK$ (Acknowledgement) – Transfer type received, ready to start. Can be sent multiple times, in different stages of communication. Sent by both the client and the server.
    - $MSGT\_DATA$ (Data) – Data transfer. Indicates that data is being sent.[284] Errors are sent as data messages.
- **Protocol** (1 byte): This protocol is used to indicate the type of transfer. This field is mandatory for all messages and shouldn't change during the transfer.
    - $PROTOCOL\_IPV4$ (IPv4) – This indicates that the transfer is either TCP or UDP in IPv4.
    - $PROTOCOL\_IPV6$ (IPv6) – This indicates that the transfer is either TCP or UDP in IPv6.
    - $PROTOCOL\_UNIX$ (UDS) – This indicates that the transfer is either stream or datagram socket in Unix domain. Can only work on the same machine, as it uses the file system.
    - $PROTOCOL\_MMAP$ (MMAP) – This indicates that the transfer is shared memory. Can only work on the same machine, as it uses the file system.
    - $PROTOCOL\_PIPE$ (PIPE) – This indicates that the transfer is pipe. Can only work in the same machine, as it uses the file system.
- **Param** (1 byte): This parameter is used to indicate the type of the transfer, or the parameter itself. This field is mandatory for all messages and shouldn't change during the transfer.

[284] Statistics, file name, address, etc.

- $PARAM\_TCP$ (TCP) – This indicates that the transfer is TCP. Only valid for IPv4 and IPv6 protocols.
- $PARAM\_UDP$ (UDP) – This indicates that the transfer is UDP. Only valid for IPv4 and IPv6 protocols.
- $PARAM\_STREAM$ (Stream) – This indicates that the transfer is stream socket. Only valid for Unix protocol.
- $PARAM\_DGRAM$ (Datagram) – This indicates that the transfer is datagram socket. Only valid for Unix protocol.
- $PARAM\_FILE$ (File) – This indicates that the transfer is file. Only valid for mmap and pipe protocols.

- **Error Code** (1 byte): This error code is used to indicate the type of error. The error code also provides a short description of the error, as a string payload in the message. In case of an error, both parties should close the connection immediately, preventing any further communication.
  - $ERRC\_SUCCESS$ (Success) – No error, normal operation. This indicates that the operation was successful. This is the only error code that doesn't have a string payload.
  - $ERRC\_SOCKET$ (Socket error) – This indicates that an error occurred in the socket itself.
  - $ERRC\_SEND$ (Send error) – Error in $send()$ or $sendto()$.
  - $ERRC\_RECV$ (Receive error) – Error in $recv()$ or $recvfrom()$.
  - $ERRC\_MMAP$ (Memory map error) – Error in MMAP.
  - $ERRC\_PIPE$ (Pipe error) – Error in Pipe. This indicates that an error occurred in one of the $pipe()$ functions.
  - $ERRC\_ALLOC$ (Memory allocation error) – Error in memory allocation. This indicates that an error occurred in one of the memory allocation functions.[285]
- **Payload Size** (4 bytes): The size of the payload. For MSGT_INIT, there isn't a payload, so this field is used to indicate the size of the file. For MSGT_DATA, this is the size of the payload itself. In all other cases, this should always be 0, as there is no payload.
- **Payload** (0-1024 bytes): The payload itself (statistics, file name, address, etc.) or the error message.

Total STNC packet size: 8 bytes + 0-1024 bytes = 8-1032 bytes.

<u>STNC Generated Data</u>

The generated data has a header sized 64 bytes with the following data (In HEX):

```
53544E43005350454544544553540052 4F595F53494D414E4F564943485F4C494E4F525F524F4E454E00415249454C5F4F505359535F4857335F4D4159323300
```

This header translates to the following ASCII code:

STNC[286] SPEEDTEST[287] ROY_SIMANOVICH_LINOR_RONEN[288] ARIEL_OPSYS_HW3_MAY23[289]

[285] $malloc(3)$, $calloc(3)$ or $realloc(3)$.
[286] The magic byte codes.
[287] Purpose of the data.
[288] Authors of the assignment.
[289] University, course, assignment, month, and year.

Performance Mode utilities:

```
#include "stnc.h"
#include <stdlib.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <openssl/evp.h>
uint8_t *util_generate_random_data(uint32_t size, bool quietMode) {
	uint8_t *buffer = NULL;
	buffer = (uint8_t *)calloc(size, sizeof(uint8_t));
	srand(time(NULL));
	memcpy(buffer, data_signeture, sizeof(data_signeture));
	for (uint32_t i = sizeof(data_signeture); i < size; i++)
		*(buffer + i) = ((uint32_t)rand() % 256);
	return buffer;
}
int32_t stnc_prepare_packet(uint8_t *buffer, stnc_message_type type, stnc_transfer_protocol protocol, stnc_transfer_param param, stnc_error_code error, uint32_t
size, uint8_t *data) {
	stnc_packet *packet = (stnc_packet *)buffer;
	packet->type = type;
	packet->protocol = protocol;
	packet->param = param;
	packet->error = error;
	packet->size = size;
	if (data != NULL && size > 0)
		memcpy(buffer + sizeof(stnc_packet), data, size);
	return 0;
}
int32_t stnc_send_tcp_data(int32_t socket, uint8_t *packet, bool quietMode) {
	int32_t bytesToSend = 0;
	send(socket, packet, (((stnc_packet *)packet)->type == MSGT_DATA ? sizeof(stnc_packet) + ((stnc_packet *)packet)->size:sizeof(stnc_packet)), 0);
	return bytesSent;
}
int32_t stnc_receive_tcp_data(int32_t socket, uint8_t *packet, bool quietMode) {
	ssize_t bytesReceived = recv(socket, packet, (STNC_PROTO_MAX_SIZE + sizeof(stnc_packet)), 0);
	if (bytesReceived <= 0) return -1;
	return bytesReceived;
}
char* util_md5_checksum(uint8_t *data, uint32_t size) {
	EVP_MD_CTX *mdctx;
	uint8_t *md5_digest = NULL;
	char *checksumString = NULL;
	uint32_t md5_digest_len = EVP_MD_size(EVP_md5());
	mdctx = EVP_MD_CTX_new();
	EVP_DigestInit_ex(mdctx, EVP_md5(), NULL);
	EVP_DigestUpdate(mdctx, data, size);
	md5_digest = (uint8_t *)OPENSSL_malloc(md5_digest_len);
	EVP_DigestFinal_ex(mdctx, md5_digest, &md5_digest_len);
	EVP_MD_CTX_free(mdctx);
	checksumString = (char *)calloc((md5_digest_len * 2 + 1), sizeof(int8_t));
	for (uint32_t i = 0; i < md5_digest_len; i++)
		sprintf(checksumString + (i * 2), "%02x", md5_digest[i]);
	OPENSSL_free(md5_digest);
	return checksumString;
}
```

Client main function:[290]

```
int32_t stnc_client_performance(char *ip, char *port, char *transferProtocol, char *transferParam, bool quietMode) {
	struct sockaddr_in serverAddress;
	uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
	uint8_t *data_to_send = NULL;
	int32_t chatSocket = INVALID_SOCKET;
	uint16_t portNumber = atoi(port);
	stnc_transfer_protocol protocol = stnc_get_transfer_protocol(transferProtocol);
	stnc_transfer_param param = stnc_get_transfer_param(transferParam);
	if (protocol == PROTOCOL_MMAP || protocol == PROTOCOL_PIPE) param = PARAM_FILE;
	data_to_send = util_generate_random_data(FILE_SIZE, quietMode);
	char *md5 = util_md5_checksum(data_to_send, FILE_SIZE);
	memset(&serverAddress, 0, sizeof(serverAddress));
	serverAddress.sin_family = AF_INET;
	serverAddress.sin_port = htons(portNumber);
	inet_pton(AF_INET, ip, &serverAddress.sin_addr);
	chatSocket = socket(AF_INET, SOCK_STREAM, 0);
	connect(chatSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress));
	stnc_prepare_packet(buffer, MSGT_INIT, protocol, param, ERRC_SUCCESS, FILE_SIZE, NULL);
	stnc_send_tcp_data(chatSocket, buffer, quietMode);
	stnc_receive_tcp_data(chatSocket, buffer, quietMode);
	stnc_prepare_packet(buffer, MSGT_DATA, protocol, param, ERRC_SUCCESS, (strlen(md5) + 1), (uint8_t*)md5);
	stnc_send_tcp_data(chatSocket, buffer, quietMode);
	if (protocol == PROTOCOL_MMAP || protocol == PROTOCOL_PIPE) {
		stnc_receive_tcp_data(chatSocket, buffer, quietMode);
		stnc_prepare_packet(buffer, MSGT_DATA, protocol, param, ERRC_SUCCESS, (strlen(transferParam) + 1), (uint8_t*)transferParam);
		stnc_send_tcp_data(chatSocket, buffer, quietMode);
	}
	stnc_receive_tcp_data(chatSocket, buffer, quietMode);
	int32_t ret = 0;
	switch(protocol) {
		case PROTOCOL_IPV4:
			ret = stnc_perf_client_ipv4(data_to_send, chatSocket, FILE_SIZE, ip, (portNumber + 1), param, quietMode);
			break;
		case PROTOCOL_IPV6:
			char ipv6Address[] = "::1";
			ret = stnc_perf_client_ipv6(data_to_send, chatSocket, FILE_SIZE, ipv6Address, (portNumber + 1), param, quietMode);
			break;
		case PROTOCOL_UNIX:
			ret = stnc_perf_client_unix(data_to_send, chatSocket, FILE_SIZE, STNC_UNIX_NAME, param, quietMode);
			break;
		case PROTOCOL_MMAP:
			ret = stnc_perf_client_memory(chatSocket, transferParam, data_to_send, FILE_SIZE, quietMode);
			break;
		case PROTOCOL_PIPE:
			ret = stnc_perf_client_pipe(chatSocket, transferParam, data_to_send, FILE_SIZE, quietMode);
			break;
	}
	double transferTime = (double)(endC.tv_sec - startC.tv_sec) + ((double)(endC.tv_usec - startC.tv_usec) / 1000000);
	fprintf(stdout, "Statistics data for this transfer (client side):\n");
	fprintf(stdout, "Sent total of %d bytes (%d KB, %d MB).\n", ret, (ret / 1024), (ret / (1024 * 1024)));
	fprintf(stdout, "Transfer time: %0.3lf seconds (%0.3lf ms).\n"
			"Transfer speed: %0.3lf KB/s (%0.3lf MB/s).\n",
			transferTime, transferTime * 1000, ((double)ret / 1024) / transferTime, ((double)ret / (1024 * 1024)) / transferTime);
	stnc_prepare_packet(buffer, MSGT_ACK, protocol, param, ERRC_SUCCESS, 0, NULL);
	stnc_send_tcp_data(chatSocket, buffer, quietMode);
	fprintf(stdout, "\nStatistics data for this transfer (server side):\n");
	stnc_receive_tcp_data(chatSocket, buffer, quietMode);
	stnc_print_packet_payload((stnc_packet *)buffer);
	close(chatSocket);
	free(data_to_send);
	free(md5);
	return EXIT_SUCCESS;
}
```

[290] Full code is available on GitHub.

Server main function:

```
int32_t stnc_server_performance(char *port, bool quietMode) {
	struct sockaddr_in serverAddress, clientAddress;
	uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
	char fileName[STNC_PROTO_MAX_SIZE] = { 0 };
	char *md5Hash = NULL;
	uint8_t *data_to_receive = NULL;
	stnc_packet *packetData = (stnc_packet *)buffer;
	socklen_t clientAddressLength = sizeof(clientAddress);
	uint16_t portNumber = atoi(port);
	stnc_transfer_protocol protocol = PROTOCOL_NONE;
	stnc_transfer_param param = PARAM_NONE;
	uint32_t fileSize = 0;
	double transferTime = 0.0;
	int32_t actual_received = 0, chatSocket = INVALID_SOCKET, serverSocket = INVALID_SOCKET, reuse = 1;
	memset(&serverAddress, 0, sizeof(serverAddress));
	memset(&clientAddress, 0, sizeof(clientAddress));
	serverAddress.sin_family = AF_INET;
	serverAddress.sin_port = htons(portNumber);
	serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
	serverSocket = socket(AF_INET, SOCK_STREAM, 0);
	setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
	bind(serverSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress));
	listen(serverSocket, 1);
	while (true) {
		chatSocket = accept(serverSocket, (struct sockaddr *)&clientAddress, &clientAddressLength);
		stnc_receive_tcp_data(chatSocket, buffer, quietMode);
		protocol = stnc_get_packet_protocol(buffer);
		param = stnc_get_packet_param(buffer);
		fileSize = stnc_get_packet_size(buffer);
		data_to_receive = (uint8_t *)malloc(fileSize * sizeof(uint8_t));
		stnc_prepare_packet(buffer, MSGT_ACK, protocol, param, ERRC_SUCCESS, 0, NULL);
		stnc_send_tcp_data(chatSocket, buffer, quietMode);
		char md5HashExpected[33] = { 0 };
		stnc_receive_tcp_data(chatSocket, buffer, quietMode);
		memcpy(md5HashExpected, ((char*)buffer + sizeof(stnc_packet)), 32);
		char *transferName = NULL;
		switch(protocol) {
			case PROTOCOL_IPV4:
				if (param == PARAM_TCP) transferName = "ipv4_tcp";
				else transferName = "ipv4_udp";
				break;
			case PROTOCOL_IPV6:
				if (param == PARAM_TCP) transferName = "ipv6_tcp";
				else transferName = "ipv6_udp";
				break;
			case PROTOCOL_UNIX:
				if (param == PARAM_STREAM) transferName = "uds_stream";
				else transferName = "uds_dgram";
				break;
			case PROTOCOL_MMAP:
				transferName = "mmap";
				break;
			case PROTOCOL_PIPE:
				transferName = "pipe";
				break;
		}
		if (protocol == PROTOCOL_MMAP || protocol == PROTOCOL_PIPE) {
			stnc_prepare_packet(buffer, MSGT_ACK, protocol, param, ERRC_SUCCESS, 0, NULL);
			stnc_send_tcp_data(chatSocket, buffer, quietMode);
			stnc_receive_tcp_data(chatSocket, buffer, quietMode);
			strcpy(fileName, ((char *)packetData + sizeof(stnc_packet)));
		}
		switch(protocol) {
			case PROTOCOL_IPV4:
				actual_received = stnc_perf_server_ipv4(chatSocket, data_to_receive, fileSize, (portNumber + 1), param, quietMode);
				break;
			case PROTOCOL_IPV6:
				actual_received = stnc_perf_server_ipv6(chatSocket, data_to_receive, fileSize, (portNumber + 1), param, quietMode);
				break;
			case PROTOCOL_UNIX:
				actual_received = stnc_perf_server_unix(chatSocket, data_to_receive, fileSize, STNC_UNIX_NAME, param, quietMode);
				break;
			case PROTOCOL_MMAP:
				stnc_prepare_packet(buffer, MSGT_ACK, PROTOCOL_MMAP, PARAM_FILE, ERRC_SUCCESS, 0, NULL);
				stnc_send_tcp_data(chatSocket, buffer, quietMode);
				stnc_receive_tcp_data(chatSocket, buffer, quietMode);
				actual_received = stnc_perf_server_memory(chatSocket, data_to_receive, fileSize, fileName, quietMode);
				break;
			case PROTOCOL_PIPE:
				actual_received = stnc_perf_server_pipe(chatSocket, data_to_receive, fileSize, fileName, quietMode);
				break;
		}
		md5Hash = util_md5_checksum(data_to_receive, actual_received);
		transferTime = (double)(end.tv_sec - start.tv_sec) + ((double)(end.tv_usec - start.tv_usec) / 1000000);
		char statics[STNC_PROTO_MAX_SIZE] = { 0 };
		snprintf(statics, (sizeof(statics) - 1), "Total data received: %u KB (%0.2f%%)\n"
	"Transfer time: %0.3lf seconds (%0.3lf ms)\n"
	"Transfer rate: %0.3lf KB/s (%0.3lf MB/s)\n"
	"Received data MD5 hash: %s (%s)",
	(actual_received / 1024),
	(((float)actual_received / (float)fileSize) * 100),
	transferTime, (transferTime * 1000),
	(((double)actual_received / 1024) / transferTime),
	(((double)actual_received / (1024 * 1024)) / transferTime),
	md5Hash,
	((strcmp(md5Hash, md5HashExpected) == 0) ? "MATCH" : "MISMATCH"));
		if (!quietMode) {
			fprintf(stdout, "%s\n", statics);
		} else {
			if (strcmp(md5Hash, md5HashExpected) == 0)
				fprintf(stdout, "%s,%d\n", transferName, (int)(transferTime * 1000));
			else
				fprintf(stdout, "%s,%d (error)\n", transferName, (int)(transferTime * 1000));
		}
		free(md5Hash);
		stnc_prepare_packet(buffer, MSGT_DATA, protocol, param, ERRC_SUCCESS, (strlen(statics) + 1), (uint8_t *)statics);
		stnc_send_tcp_data(chatSocket, buffer, quietMode);
		close(chatSocket);
		free(data_to_receive);
	}
	close(serverSocket);
	return EXIT_SUCCESS;
}
```

Protocol handling functions (Both server and client):[291]

$stnc\_perf\_client\_ipv4()$:

```
int32_t stnc_perf_client_ipv4(uint8_t* data, int32_t chatsocket, uint32_t filesize, char *server_ip, uint16_t server_port, stnc_transfer_param param, bool quietMode) {
        struct sockaddr_in serverAddress;
        uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
        int32_t serverSocket = INVALID_SOCKET;
        socklen_t len = sizeof(serverAddress);
        uint32_t bytesSent = 0;
        memset(&serverAddress, 0, sizeof(serverAddress));
        serverAddress.sin_family = AF_INET;
        serverAddress.sin_port = htons(server_port);
        inet_pton(AF_INET, server_ip, &serverAddress.sin_addr);
        socket(AF_INET, (param == PARAM_TCP) ? SOCK_STREAM : SOCK_DGRAM, 0);
        if (param == PARAM_TCP) {
                connect(serverSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress));
        }
        struct pollfd fds[2];
        fds[0].fd = chatsocket;
        fds[0].events = POLLIN;
        fds[1].fd = serverSocket;
        fds[1].events = POLLOUT;
        gettimeofday(&startC, NULL);
        while (bytesSent < filesize) {
                poll(fds, 2, STNC_POLL_TIMEOUT);
                if (fds[0].revents & POLLIN) {
                        stnc_receive_tcp_data(chatsocket, buffer, quietMode);
                }
                if (fds[1].revents & POLLOUT) {
                        if (param == PARAM_TCP) {
                                uint32_t bytesToSend = (((filesize - bytesSent) > CHUNK_SIZE) ? CHUNK_SIZE:(filesize - bytesSent));
                                int32_t bytes = send(serverSocket, data + bytesSent, bytesToSend, 0);
                                gettimeofday(&endC, NULL);
                                bytesSent += bytes;
                        } else {
                                uint32_t bytesToSend = (((filesize - bytesSent) > CHUNK_SIZE_UDP) ? CHUNK_SIZE_UDP:(filesize - bytesSent));
                                int32_t bytes = sendto(serverSocket, data + bytesSent, bytesToSend, 0, (struct sockaddr *)&serverAddress, len);
                                gettimeofday(&endC, NULL);
                                bytesSent += bytes;
                        }
                }
        }
        close(serverSocket);
        return bytesSent;
}
```

$stnc\_perf\_client\_ipv6()$:

```
int32_t stnc_perf_client_ipv6(uint8_t* data, int32_t chatsocket, uint32_t filesize, char *server_ip, uint16_t server_port, stnc_transfer_param param, bool quietMode) {
        struct sockaddr_in6 serverAddress;
        uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
        int32_t serverSocket = INVALID_SOCKET;
        uint32_t bytesSent = 0;
        memset(&serverAddress, 0, sizeof(serverAddress));
        serverAddress.sin6_family = AF_INET6;
        serverAddress.sin6_port = htons(server_port);
        inet_pton(AF_INET6, server_ip, &serverAddress.sin6_addr);
        socket(AF_INET6, (param == PARAM_TCP) ? SOCK_STREAM : SOCK_DGRAM, 0);
        if (param == PARAM_TCP) {
                connect(serverSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress));
        }
        struct pollfd fds[2];
        fds[0].fd = chatsocket;
        fds[0].events = POLLIN;
        fds[1].fd = serverSocket;
        fds[1].events = POLLOUT;
        gettimeofday(&startC, NULL);
        while (bytesSent < filesize) {
                poll(fds, 2, STNC_POLL_TIMEOUT);
                if (fds[0].revents & POLLIN) {
                        stnc_receive_tcp_data(chatsocket, buffer, quietMode);
                }
                if (fds[1].revents & POLLOUT) {
                        if (param == PARAM_TCP)
                        {
                                uint32_t bytesToSend = (((filesize - bytesSent) > CHUNK_SIZE) ? CHUNK_SIZE:(filesize - bytesSent));
                                int32_t bytes = send(serverSocket, data + bytesSent, bytesToSend, 0);
                                gettimeofday(&endC, NULL);
                                bytesSent += bytes;
                        } else {
                                uint32_t bytesToSend = (((filesize - bytesSent) > CHUNK_SIZE_UDP) ? CHUNK_SIZE_UDP:(filesize - bytesSent));
                                int32_t bytes = sendto(serverSocket, data + bytesSent, bytesToSend, 0, (struct sockaddr *)&serverAddress, sizeof(serverAddress));
                                gettimeofday(&endC, NULL);
                                bytesSent += bytes;
                        }
                }
        }
        close(serverSocket);
        return bytesSent;
}
```

[291] For simplicity of the code, it doesn't include any error check. Full code avaliable at GitHub.

$stnc\_perf\_client\_unix()$:

```
int32_t stnc_perf_client_unix(uint8_t* data, int32_t chatsocket, uint32_t filesize, char *server_uds_path, stnc_transfer_param param, bool quietMode) {
        struct sockaddr_un serverAddress;
        uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
        int32_t serverSocket = INVALID_SOCKET;
        socklen_t len = strlen(server_uds_path) + sizeof(serverAddress.sun_family);
        uint32_t bytesSent = 0;
        memset(&serverAddress, 0, sizeof(serverAddress));
        serverAddress.sun_family = AF_UNIX;
        strcpy(serverAddress.sun_path, server_uds_path);
        serverSocket = socket(AF_UNIX, (param == PARAM_STREAM) ? SOCK_STREAM : SOCK_DGRAM, 0);
        if (param == PARAM_STREAM) {
                connect(serverSocket, (struct sockaddr *)&serverAddress, len);
        }
        struct pollfd fds[2];
        fds[0].fd = chatsocket;
        fds[0].events = POLLIN;
        fds[1].fd = serverSocket;
        fds[1].events = POLLOUT;
        gettimeofday(&startC, NULL);
        while (bytesSent < filesize) {
                poll(fds, 2, STNC_POLL_TIMEOUT);
                if (fds[0].revents & POLLIN) {
                        stnc_receive_tcp_data(chatsocket, buffer, quietMode);
                }
                if (fds[1].revents & POLLOUT) {
                        if (param == PARAM_STREAM)
                        {
                                uint32_t bytesToSend = (((filesize - bytesSent) > CHUNK_SIZE) ? CHUNK_SIZE:(filesize - bytesSent));
                                int32_t bytes = send(serverSocket, data + bytesSent, bytesToSend, 0);
                                gettimeofday(&endC, NULL);
                                bytesSent += bytes;
                        } else {
                                uint32_t bytesToSend = (((filesize - bytesSent) > CHUNK_SIZE_UDP) ? CHUNK_SIZE_UDP:(filesize - bytesSent));
                                int32_t bytes = sendto(serverSocket, data + bytesSent, bytesToSend, 0, (struct sockaddr *)&serverAddress, len);
                                gettimeofday(&endC, NULL);
                                bytesSent += bytes;
                        }
                }
        }
        close(serverSocket);
        return bytesSent;
}
```

$stnc\_perf\_client\_memory()$:

```
int32_t stnc_perf_client_memory(int32_t chatsocket, char *file_name, uint8_t *dataToSend, uint32_t filesize, bool quietMode) {
        uint8_t *data = MAP_FAILED;
        uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
        int32_t fd = INVALID_SOCKET;
        FILE *fp = NULL;
        unlink(file_name);
        fp = fopen(file_name, "w+");
        fd = fileno(fp);
        ftruncate(fd, sizeof(uint32_t) + filesize);
        data = mmap(NULL, sizeof(uint32_t) + filesize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        uint8_t *dataPtr = data + sizeof(uint32_t);
        uint32_t *bytesWritten = (uint32_t *)data;
        uint32_t bytesSent = 0;
        *bytesWritten = 0;
        struct pollfd fds[2];
        fds[0].fd = chatsocket;
        fds[0].events = POLLIN;
        fds[1].fd = fd;
        fds[1].events = POLLOUT;
        stnc_prepare_packet(buffer, MSGT_ACK, PROTOCOL_MMAP, PARAM_FILE, ERRC_SUCCESS, 0, NULL);
        stnc_send_tcp_data(chatsocket, buffer, quietMode);
        gettimeofday(&startC, NULL);
        while (bytesSent < filesize) {
                poll(fds, 2, STNC_POLL_TIMEOUT);
                if (fds[0].revents & POLLIN) {
                        stnc_receive_tcp_data(chatsocket, buffer, quietMode);
                } else if (fds[1].revents & POLLOUT) {
                        uint32_t bytesToSend = (((filesize - bytesSent) > CHUNK_SIZE) ? CHUNK_SIZE:(filesize - bytesSent));
                        memcpy(dataPtr, dataToSend, bytesToSend);
                        gettimeofday(&endC, NULL);
                        dataToSend += bytesToSend;
                        dataPtr += bytesToSend;
                        bytesSent += bytesToSend;
                        *bytesWritten = bytesSent;
                }
        }
        munmap(data, sizeof(uint32_t) + filesize);
        fclose(fp);
        return bytesSent;
}
```

$stnc\_perf\_client\_pipe()$:

```
int32_t stnc_perf_client_pipe(int32_t chatsocket, char *fifo_name, uint8_t *dataToSend, uint32_t filesize, bool quietMode) {
        uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
        int32_t fd = INVALID_SOCKET;
        uint32_t bytesSent = 0;
        mkfifo(fifo_name, 0644);
        fd = open(fifo_name, O_WRONLY);
        struct pollfd fds[2];
        fds[0].fd = chatsocket;
        fds[0].events = POLLIN;
        fds[1].fd = fd;
        fds[1].events = POLLOUT;
        gettimeofday(&startC, NULL);
        while (bytesSent < filesize) {
                poll(fds, 2, STNC_POLL_TIMEOUT);
                if (fds[0].revents & POLLIN) {
                        stnc_receive_tcp_data(chatsocket, buffer, quietMode);
                } else if (fds[1].revents & POLLOUT) {
                        uint32_t bytesToSend = (((filesize - bytesSent) > CHUNK_SIZE) ? CHUNK_SIZE:(filesize - bytesSent));
                        write(fd, dataToSend + bytesSent, bytesToSend);
                        gettimeofday(&endC, NULL);
                        bytesSent += bytesToSend;
                }
        }
        close(fd);
        return bytesSent;
}
```

$stnc\_perf\_server\_pipe()$:

```
int32_t stnc_perf_server_pipe(int32_t chatsocket, uint8_t* data, uint32_t filesize, char *file_name, bool quietMode) {
        uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
        stnc_prepare_packet(buffer, MSGT_ACK, PROTOCOL_PIPE, PARAM_FILE, ERRC_SUCCESS, 0, NULL);
        stnc_send_tcp_data(chatsocket, buffer, quietMode);
        sleep(1);
        int32_t fd = open(file_name, O_RDONLY);
        uint32_t bytesReceived = 0;
        struct pollfd fds[2];
        fds[0].fd = chatsocket;
        fds[0].events = POLLIN;
        fds[1].fd = fd;
        fds[1].events = POLLIN;
        gettimeofday(&start, NULL);
        while (bytesReceived < filesize) {
                int32_t ret = poll(fds, 2, STNC_POLL_TIMEOUT);
                if (ret == 0) {
                        break;
                } else if (fds[0].revents & POLLIN) {
                        stnc_receive_tcp_data(chatsocket, buffer, quietMode);
                } else if (fds[1].revents & POLLIN) {
                        uint32_t bytesToReceived = ((filesize - bytesReceived) > CHUNK_SIZE) ? CHUNK_SIZE:(filesize - bytesReceived);
                        read(fd, data + bytesReceived, bytesToReceived);
                        gettimeofday(&end, NULL);
                        bytesReceived += bytesToReceived;
                }
        }
        close(fd);
        // Clean up, remove the file, as it's no longer needed.
        unlink(file_name);
        return bytesReceived;
}
```

***stnc_perf_server_memory*()**:

```
int32_t stnc_perf_server_memory(int32_t chatsocket, uint8_t* data, uint32_t filesize, char *file_name, bool quietMode) {
	uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
	uint8_t *dataToReceive = MAP_FAILED;
	int32_t fd = INVALID_SOCKET;
	FILE* fp = fopen(file_name, "r");
	fd = fileno(fp);
	dataToReceive = mmap(NULL, sizeof(uint32_t) + filesize, PROT_READ, MAP_SHARED, fd, 0);
	uint8_t *dataToReceive_tmp = dataToReceive + sizeof(uint32_t);
	uint32_t *bytesSendFromMemory = (uint32_t *)dataToReceive;
	uint32_t bytesReceived = 0;
	struct pollfd fds[2];
	fds[0].fd = chatsocket;
	fds[0].events = POLLIN;
	fds[1].fd = fd;
	fds[1].events = POLLIN;
	gettimeofday(&start, NULL);
	while (bytesReceived < filesize) {
		int32_t ret = poll(fds, 2, STNC_POLL_TIMEOUT);
		if (ret == 0) {
			break;
		} else if (fds[0].revents & POLLIN) {
			stnc_receive_tcp_data(chatsocket, buffer, quietMode);
		} else if (fds[1].revents & POLLIN) {
			uint32_t bytesToReceived = ((*bytesSendFromMemory - bytesReceived) > CHUNK_SIZE) ? CHUNK_SIZE:(*bytesSendFromMemory - bytesReceived);
			memcpy(data, dataToReceive_tmp, bytesToReceived);
			gettimeofday(&end, NULL);
			data += bytesToReceived;
			dataToReceive_tmp += bytesToReceived;
			bytesReceived += bytesToReceived;
		}
	}
	munmap(dataToReceive, sizeof(uint32_t) + filesize);
	fclose(fp);
	// Clean up, remove the file, as it's no longer needed.
	unlink(file_name);
	return bytesReceived;
}
```

$stnc\_perf\_server\_unix()$:

```
int32_t stnc_perf_server_unix(int32_t chatsocket, uint8_t* data, uint32_t filesize, char *server_uds_path, stnc_transfer_param param, bool quietMode) {
	uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
	struct sockaddr_un serverAddress, clientAddress;
	uint32_t bytesReceived = 0;
	socklen_t clientAddressLength = sizeof(clientAddress);
	int serverSocket = INVALID_SOCKET;
	memset(&serverAddress, 0, sizeof(struct sockaddr_un));
	memset(&clientAddress, 0, sizeof(struct sockaddr_un));
	serverAddress.sun_family = AF_UNIX;
	strcpy(serverAddress.sun_path, server_uds_path);
	// Clean up the socket file if it already exists.
	unlink(server_uds_path);
	int len = strlen(serverAddress.sun_path) + sizeof(serverAddress.sun_family);
	serverSocket = socket(AF_UNIX, (param == PARAM_STREAM ? SOCK_STREAM : SOCK_DGRAM), 0);
	bind(serverSocket, (struct sockaddr *)&serverAddress, len);
	if (param == PARAM_STREAM) {
		listen(serverSocket, 1);
		stnc_prepare_packet(buffer, MSGT_ACK, PROTOCOL_IPV4, param, ERRC_SUCCESS, 0, NULL);
		stnc_send_tcp_data(chatsocket, buffer, quietMode);
		int32_t clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddress, &clientAddressLength);
		close(serverSocket);
		struct pollfd fds[2];
		fds[0].fd = chatsocket;
		fds[0].events = POLLIN;
		fds[1].fd = clientSocket;
		fds[1].events = POLLIN;
		gettimeofday(&start, NULL);
		while (bytesReceived < filesize) {
			int32_t ret = poll(fds, 2, STNC_POLL_TIMEOUT);
			if (ret == 0) {
				break;
			} else if (fds[0].revents & POLLIN) {
				stnc_receive_tcp_data(chatsocket, buffer, quietMode);
			} else if (fds[1].revents & POLLIN) {
				uint32_t bytesToReceive = (((filesize - bytesReceived) > CHUNK_SIZE) ? CHUNK_SIZE:(filesize - bytesReceived));
				int32_t bytes = 0;
				bytes = recv(clientSocket, data + bytesReceived, bytesToReceive, 0);
				gettimeofday(&end, NULL);
				bytesReceived += (uint32_t)bytes;
			}
		}
		close(clientSocket);
	} else {
		struct pollfd fds[2];
		fds[0].fd = chatsocket;
		fds[0].events = POLLIN;
		fds[1].fd = serverSocket;
		fds[1].events = POLLIN;
		stnc_prepare_packet(buffer, MSGT_ACK, PROTOCOL_IPV4, param, ERRC_SUCCESS, 0, NULL);
		stnc_send_tcp_data(chatsocket, buffer, quietMode);
		gettimeofday(&start, NULL);
		while (bytesReceived < filesize) {
			int32_t ret = poll(fds, 2, STNC_POLL_TIMEOUT);
			if (ret == 0) {
				break;
			} else if (ret > 0) {
				if (fds[0].revents & POLLIN) {
					stnc_receive_tcp_data(chatsocket, buffer, quietMode);
				} else if (fds[1].revents & POLLIN) {
					uint32_t bytesToReceive = (((filesize - bytesReceived) > CHUNK_SIZE_UDP) ? CHUNK_SIZE_UDP:(filesize - bytesReceived));
					int32_t bytes = 0;
					bytes = recvfrom(serverSocket, data + bytesReceived, bytesToReceive, 0, (struct sockaddr *)&clientAddress, &clientAddressLength);
					gettimeofday(&end, NULL);
					bytesReceived += (uint32_t)bytes;
				}
			}
		}
		close(serverSocket);
	}
	// Cleanup
	unlink(server_uds_path);
	return bytesReceived;
}
```

***stnc_perf_server_ipv6*()**:

```
int32_t stnc_perf_server_ipv6(int32_t chatsocket, uint8_t* data, uint32_t filesize, uint16_t server_port, stnc_transfer_param param, bool quietMode) {
	uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
	struct sockaddr_in6 serverAddress, clientAddress;
	socklen_t len = sizeof(clientAddress);
	uint32_t bytesReceived = 0;
	int32_t serverSocket = INVALID_SOCKET, reuse = 1;
	serverSocket = socket(AF_INET6, (param == PARAM_TCP ? SOCK_STREAM:SOCK_DGRAM), 0);
	setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
	memset(&serverAddress, 0, sizeof(serverAddress));
	memset(&clientAddress, 0, sizeof(clientAddress));
	serverAddress.sin6_family = AF_INET6;
	serverAddress.sin6_port = htons(server_port);
	serverAddress.sin6_addr = in6addr_any;
	bind(serverSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress));
	if (param == PARAM_TCP) {
		listen(serverSocket, 1);
		stnc_prepare_packet(buffer, MSGT_ACK, PROTOCOL_IPV4, param, ERRC_SUCCESS, 0, NULL);
		stnc_send_tcp_data(chatsocket, buffer, quietMode);
		int32_t clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddress, &len);
		close(serverSocket);
		struct pollfd fds[2];
		fds[0].fd = chatsocket;
		fds[0].events = POLLIN;
		fds[1].fd = clientSocket;
		fds[1].events = POLLIN;
		gettimeofday(&start, NULL);
		while (bytesReceived < filesize) {
			int32_t ret = poll(fds, 2, STNC_POLL_TIMEOUT);
			if (ret == 0) {
				break;
			} else if (fds[0].revents & POLLIN) {
				stnc_receive_tcp_data(chatsocket, buffer, quietMode);
			} else if (fds[1].revents & POLLIN) {
				uint32_t bytesToReceive = (((filesize - bytesReceived) > CHUNK_SIZE) ? CHUNK_SIZE:(filesize - bytesReceived));
				int32_t bytes = 0;
				bytes = recv(clientSocket, data + bytesReceived, bytesToReceive, 0);
				gettimeofday(&end, NULL);
				bytesReceived += (uint32_t)bytes;
			}
		}
		close(clientSocket);
	} else {
		struct pollfd fds[2];
		fds[0].fd = chatsocket;
		fds[0].events = POLLIN;
		fds[1].fd = serverSocket;
		fds[1].events = POLLIN;
		stnc_prepare_packet(buffer, MSGT_ACK, PROTOCOL_IPV4, param, ERRC_SUCCESS, 0, NULL);
		stnc_send_tcp_data(chatsocket, buffer, quietMode);
		gettimeofday(&start, NULL);
		while (bytesReceived < filesize) {
			int32_t ret = poll(fds, 2, STNC_POLL_TIMEOUT);
			if (ret == 0) {
				break;
			} else if (ret > 0) {
				if (fds[0].revents & POLLIN) {
					stnc_receive_tcp_data(chatsocket, buffer, quietMode);
				} else if (fds[1].revents & POLLIN) {
					uint32_t bytesToReceive = (((filesize - bytesReceived) > CHUNK_SIZE_UDP) ? CHUNK_SIZE_UDP:(filesize - bytesReceived));
					int32_t bytes = 0;
					bytes = recvfrom(serverSocket, data + bytesReceived, bytesToReceive, 0, (struct sockaddr *)&clientAddress, &len);
					gettimeofday(&end, NULL);
					bytesReceived += (uint32_t)bytes;
				}
			}
		}
		close(serverSocket);
	}
	return bytesReceived;
}
```

***stnc_perf_server_ipv4*()**:

```
int32_t stnc_perf_server_ipv4(int32_t chatsocket, uint8_t* data, uint32_t filesize, uint16_t server_port, stnc_transfer_param param, bool quietMode) {
	uint8_t buffer[STNC_PROTO_MAX_SIZE] = { 0 };
	struct sockaddr_in serverAddress, clientAddress;
	socklen_t len = sizeof(clientAddress);
	uint32_t bytesReceived = 0;
	int32_t serverSocket = INVALID_SOCKET, reuse = 1;
	serverSocket = socket(AF_INET, (param == PARAM_TCP ? SOCK_STREAM:SOCK_DGRAM), 0);
	setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
	memset(&serverAddress, 0, sizeof(serverAddress));
	memset(&clientAddress, 0, sizeof(clientAddress));
	serverAddress.sin_family = AF_INET;
	serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
	serverAddress.sin_port = htons(server_port);
	bind(serverSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress));
	if (param == PARAM_TCP) {
		listen(serverSocket, 1);
		stnc_prepare_packet(buffer, MSGT_ACK, PROTOCOL_IPV4, param, ERRC_SUCCESS, 0, NULL);
		stnc_send_tcp_data(chatsocket, buffer, quietMode);
		int32_t clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddress, &len);
		close(serverSocket);
		struct pollfd fds[2];
		fds[0].fd = chatsocket;
		fds[0].events = POLLIN;
		fds[1].fd = clientSocket;
		fds[1].events = POLLIN;
		gettimeofday(&start, NULL);
		while (bytesReceived < filesize) {
			int32_t ret = poll(fds, 2, STNC_POLL_TIMEOUT);
			if (ret == 0) {
				break;
			} else if (fds[0].revents & POLLIN) {
				stnc_receive_tcp_data(chatsocket, buffer, quietMode);
			} else if (fds[1].revents & POLLIN) {
				uint32_t bytesToReceive = (((filesize - bytesReceived) > CHUNK_SIZE) ? CHUNK_SIZE:(filesize - bytesReceived));
				int32_t bytes = 0;
				bytes = recv(clientSocket, data + bytesReceived, bytesToReceive, 0);
				gettimeofday(&end, NULL);
				bytesReceived += (uint32_t)bytes;
			}
		}
		close(clientSocket);
	} else {
		struct pollfd fds[2];
		fds[0].fd = chatsocket;
		fds[0].events = POLLIN;
		fds[1].fd = serverSocket;
		fds[1].events = POLLIN;
		stnc_prepare_packet(buffer, MSGT_ACK, PROTOCOL_IPV4, param, ERRC_SUCCESS, 0, NULL);
		stnc_send_tcp_data(chatsocket, buffer, quietMode);
		gettimeofday(&start, NULL);
		while (bytesReceived < filesize) {
			int32_t ret = poll(fds, 2, STNC_POLL_TIMEOUT);
			if (ret == 0) {
				break;
			} else if (ret > 0) {
				if (fds[0].revents & POLLIN)
				{
					stnc_receive_tcp_data(chatsocket, buffer, quietMode);
				} else if (fds[1].revents & POLLIN) {
					uint32_t bytesToReceive = (((filesize - bytesReceived) > CHUNK_SIZE_UDP) ? CHUNK_SIZE_UDP:(filesize - bytesReceived));
					int32_t bytes = 0;
					bytes = recvfrom(serverSocket, data + bytesReceived, bytesToReceive, 0, (struct sockaddr *)&clientAddress, &len);
					gettimeofday(&end, NULL);
					bytesReceived += (uint32_t)bytes;
				}
			}
		}
		close(serverSocket);
	}
	return bytesReceived;
}
```

Assignment 4

Context

The goal of this task is to implement a chat that supports unlimited number of clients, while using the Reactor design pattern and using the $poll(2)$ or $select(2)$ API system calls.[292]

<u>The Reactor library</u>:

In this section you'll implement the core of the Reactor design pattern, meaning a mechanism that will receive file descriptors of clients and a pointer to a function to execute when the file descriptor is "hot". The Reactor will support handling a large amount of file descriptors and it'll run in a single thread. If there isn't any input, the reactor will wait until one of the file descriptors receives a message and executes the correct function. The library name will be $st\_reactor.so$.[293]

According to the Reactor design pattern, you should:

- Implement a data structure that will map between a file descriptor and the function that handles it.
- Implement the selector machine, that's responsible by $poll(2)$ or $select(2)$ to listen to all the file descriptors and execute the function that is related to this file descriptor.
- Because the selector listens to multiple file descriptors, there could be a situation where more than 1 of them would be "hot", in that case the order of handling them is irrelevant.

You could define a struct that saves in it the private fields of the reactor, as you wish.

The API of the library will look like this:

- $void * \ createReactor()$ – Creates a Reactor object. Returns a pointer to the created Reactor struct that will be passed to the following functions. When the Reactor is created, it doesn't start to run immediately, but all the data fields will be allocated.
- $void\ stopReactor(void\ * this)$ – Stops the Reactor if it's running. Otherwise, it doesn't do anything.
- $void\ startReactor(void\ * this)$ – Starts the Reactor thread. The thread will live in a busy waiting loop and in practice will be called $poll(2)$ or $select(2)$.
- $void\ addFd(void\ * this, int\ fd, handler\_t\ handler)$ - Adds a file descriptor and the function that handles it to the list of the Reactor.[294]
- $void\ WaitFor(void\ * this)$ – The calling thread will wait, by calling the $pthread\_join(3)$ library function, until the Reactor's thread will end.

[292] The explanation for the APIs and code examples are avaliable in chapter 7 of Beej's Guide to Network Programming.
[293] The Reactor should only support reading, there is no need for writing or exception handling.
[294] The $handler\_t$ field is a typedef of a pointer to a function that handles the file descriptor when its "hot". You can implement the signature of the function as you wish.

<u>The wrapper of the library - The application</u>:

The server will work exactly as Beej's chat.[295] The client of Beej should be compatible with the server with no issues.[296] When the thread of the Reactor runs, the main thread will wait for him.

Usage: $./react\_server$[297]

[295] Like $selectserver$.
[296] For any question about it, like in which port we should run the server, you should configure it the same as Beej's server.
[297] No need for arguments, same as Beej's.

Solution

***reactor.h*:**

```
typedef struct _reactor_node reactor_node, *reactor_node_ptr;
typedef struct _reactor_t reactor_t, *reactor_t_ptr;
typedef struct pollfd pollfd_t, *pollfd_t_ptr;
struct _reactor_node
{
        int fd;
        union _hdlr_func_union
        {
                handler_t handler;
                void *handler_ptr;
        } hdlr;
        reactor_node_ptr next;
};
struct _reactor_t
{

        pthread_t thread;
        reactor_node_ptr head;
        pollfd_t_ptr fds;
        bool running;
};
```

Reactor node struct explanation:

- fd – The file descriptor.[298]
- $handler$ – The file descriptor's handler.[299]
- $handler\_ptr$ – A pointer to the handler function.[300]
- $next$ – The next node in the linked list.[301]

Reactor object struct explanation:

- $thread$ – The thread in which the reactor is running. The thread is created in $startReactor()$ and deleted in $stopReactor()$. The thread function is $reactorRun()$.
- $head$ – The first node in the linked list.
- fds – A pointer to an array of pollfd structures. The array is allocated and freed in $reactorRun()$. The array is used in $reactorRun()$ to call poll().
- $running$ – A Boolean value indicating whether the reactor is running. The value is set to true in $startReactor()$ and to false in $stopReactor()$.

[298] The first node is always the listening socket.
[299] The first node is always the listening socket, and its handler is always to accept a new connection and add it to the reactor.
[300] This is a generic pointer, and it's used to print the handler's address.
[301] For the last node, this is NULL.

st_reactor.c:[302]

```
#include "reactor.h"
#include <arpa/inet.h>
#include <netinet/in.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

void *reactorRun(void *react) {
	reactor_t_ptr reactor = (reactor_t_ptr)react;
	while (reactor->running) {
		size_t size = 0, i = 0;
		reactor_node_ptr curr = reactor->head;
		while (curr != NULL) {
			size++;
			curr = curr->next;
		}
		curr = reactor->head;
		reactor->fds = (pollfd_t_ptr)calloc(size, sizeof(pollfd_t));
		while (curr != NULL) {
			(*(reactor->fds + i)).fd = curr->fd;
			(*(reactor->fds + i)).events = POLLIN;

			curr = curr->next;
			i++;
		}
		poll(reactor->fds, i, POLL_TIMEOUT);
		for (i = 0; i < size; ++i) {
			if ((*(reactor->fds + i)).revents & POLLIN) {
				reactor_node_ptr curr = reactor->head;
				for (unsigned int j = 0; j < i; ++j) curr = curr->next;
				void *handler_ret = curr->hdlr.handler((*(reactor->fds + i)).fd, reactor);
				if (handler_ret == NULL && (*(reactor->fds + i)).fd != reactor->head->fd) {
					reactor_node_ptr curr_node = reactor->head;
					reactor_node_ptr prev_node = NULL;
					while (curr_node != NULL && curr_node->fd != (*(reactor->fds + i)).fd) {
						prev_node = curr_node;
						curr_node = curr_node->next;
					}
					prev_node->next = curr_node->next;
					free(curr_node);
				}
				continue;
			}
		}
		free(reactor->fds);
		reactor->fds = NULL;
	}
	return reactor;
}
void *createReactor() {
	reactor_t_ptr react = NULL;
	react = (reactor_t_ptr)malloc(sizeof(reactor_t))) == NULL;
	react->thread = 0;
	react->head = NULL;
	react->fds = NULL;
	react->running = false;
	fprintf(stdout, "%s Reactor created.\n", C_PREFIX_INFO);
	return react;
}
void startReactor(void *react) {
	reactor_t_ptr reactor = (reactor_t_ptr)react;
	if (reactor->running) return;
	reactor->running = true;
	pthread_create(&reactor->thread, NULL, reactorRun, react);
}
void stopReactor(void *react) {
	reactor_t_ptr reactor = (reactor_t_ptr)react;
	if (!reactor->running) return;
	reactor->running = false;
	pthread_cancel(reactor->thread);
	pthread_join(reactor->thread, &ret);
	if (reactor->fds != NULL) {
		free(reactor->fds);
		reactor->fds = NULL;
	}
	reactor->thread = 0;
}
void addFd(void *react, int fd, handler_t handler) {
	reactor_t_ptr reactor = (reactor_t_ptr)react;
	reactor_node_ptr node = (reactor_node_ptr)malloc(sizeof(reactor_node));
	node->fd = fd;
	node->hdlr.handler = handler;
	node->next = NULL;

	if (reactor->head == NULL) {
		reactor->head = node;
	} else {
		reactor_node_ptr curr = reactor->head;
		while (curr->next != NULL) curr = curr->next;
		curr->next = node;
	}
}
void WaitFor(void *react) {
	reactor_t_ptr reactor = (reactor_t_ptr)react;
	void *ret = NULL;
	if (!reactor->running) return;
	pthread_join(reactor->thread, &ret);
}
```

[302] This piece of code doesn't have error checking for simplicity. Full code is available at GitHub.

react server.c:[303]

```
#include "reactor.h"
#include <arpa/inet.h>
#include <netinet/in.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
	struct sockaddr_in server_addr = {
		.sin_family = AF_INET,
		.sin_port = htons(SERVER_PORT),
		.sin_addr.s_addr = INADDR_ANY
	};
	int server_fd = -1, reuse = 1;
	fprintf(stdout, "Starting server...\n");
	server_fd = socket(AF_INET, SOCK_STREAM, 0);
	setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(int));
	bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
	listen(server_fd, MAX_QUEUE);
	fprintf(stdout, "Server started successfully.\n");
	void* reactor = createReactor();
	fprintf(stdout, "Adding server socket to reactor...\n");
	addFd(reactor, server_fd, server_handler);
	fprintf(stdout, "Server socket added to reactor successfully.\n");
	startReactor(reactor);
	WaitFor(reactor);
	return EXIT_SUCCESS;
}
void *client_handler(int fd, void *react) {
	char *buf = (char *)calloc(MAX_BUFFER, sizeof(char));
	int bytes_read = recv(fd, buf, MAX_BUFFER, 0);
	if (bytes_read <= 0)
	{
		fprintf(stdout, "Client %d disconnected.\n", fd);
		free(buf);
		close(fd);
		return NULL;
	}
	if (bytes_read < MAX_BUFFER)
		*(buf + bytes_read) = '\0';

	else
		*(buf + MAX_BUFFER - 1) = '\0';
	fprintf(stdout, "Client %d: %s\n", fd, buf);
	free(buf);
	return react;
}
void *server_handler(int fd, void *react) {
	struct sockaddr_in client_addr;
	socklen_t client_len = sizeof(client_addr);
	reactor_t_ptr reactor = (reactor_t_ptr)react;
	int client_fd = accept(fd, (struct sockaddr *)&client_addr, &client_len);
	fprintf(stdout, "Client %s:%d connected, Reference ID: %d\n", inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port), client_fd);
	addFd(reactor, client_fd, client_handler);
	client_count++;
	return react;
}
```

[303] This piece of code doesn't have error checking for simplicity. Full code is available at GitHub.

Assignment 5

Context

Part A:

Implement a function that receives an unsigned integer and checks if the given integer is a prime number.[304] The function will return 0 if the number isn't prime, otherwise it will return another value.

Part B:

Implement a queue in a multi-threaded environment. The queue should use mutex lock and allowing to wait for an item to be dequeued from an empty queue without busy waiting loop.[305] The queue will hold $void *$ objects.

Part C:

Implement an Active Object that supports the following functions:

- $void * CreateActiveObject(handler\_t\ handler)$ – A function that will allocate memory for an Active Object and will run its thread. Moreover, the function will allocate memory for a blocking queue and will receive a pointer to a function that will be executed for every object in the queue.
  The busy waiting loop of the Active Object will look something like this:

```
while (task = this->queue->dequeue())
        this->func(task);
```

- $void * getQueue(void * this)$ – A function that will return a pointer to the Active Object's queue. We can also use this function to enqueue an object in the queue.
- $void\ stop(void * this)$ – A function that will stop an Active Object. The function will also will cleanup all the memory allocations that were created by the Active Object.

Part D:

Use the functions that you built to write a new program.

Write a program named $st\_pipeline$ that will receive one or two arguments, the first one is called N, which represents the number of tasks and the second one is called $seed$, which represents a random seed.[306] [307]

---

[304] Only odd numbers, up to the square root. There is no need to implement the Miller–Rabin primality test or any other smart test.
[305] For example, by using condition.
[306] If the second argument isn't provided, you can generate a seed with the $time(2)$ system call.
[307] Please note that using the same random seed will generate the same sequence of pseudo-random numbers, and thus enabling automatic check.

Build a pipeline[308] by following the instructions:

- **Producer (Active Object 1)** – Generates random numbers and sends them to the queue of the first consumer.
- **First Consumer (Active Object 2)** – Receives numbers from the producer, checks if they are prime numbers, adds 11 to them and sends them to the queue of the second consumer.
- **Second Consumer (Active Object 3)** – Receives numbers from the first consumer, checks if they are prime numbers, subtracts 13 from them and sends them to the queue of the third consumer.
- **Third Consumer (Active Object 4)** – Receives numbers from the second consumer, checks if they are prime numbers, adds 2 to them and prints them. The printed number should be the same as the number generated by the producer.

After the program finishes with the N tasks, it will exit.

Run example:

```
./st_pipeline 2
174013
false
174024
false
174011
false
174013
289961
false
289972
false
289959
false
289961
```

[308] **Reminder:** A pipeline is a collection of Active Objects.

Solution

Task API

The Task API supports the following operations:

- $PTask\ createTask(unsigned\ int\ num\_of\_tasks, unsigned\ int\ \_data)$ – Creates a new task with the given data (num_of_tasks and $\_data$), allocates memory for it and returns a pointer to the task.
- $void\ taskDestroy(PTask\ task)$ – Destroys a task - frees the memory allocated for the task.

The Task struct[309] has the following fields:

- $unsigned\ int\ num\_of\_tasks$ – The number of tasks that the task has been through.
- $unsigned\ int\ \_data$ – The data of the task. In this assignment, the data is a number: It's used by the producer to generate a random number, and by the consumers to check if the number is prime, and to add or subtract a number from it.

Queue API

The Queue API,[310] which is thread safe, supports the following operations:

- $PQueue\ queueCreate()$ – Creates a new queue and returns a pointer to the queue.
- $void\ queueDestroy(PQueue\ queue)$ – Destroys a queue, destroys all the nodes in the queue and frees the memory allocated for the queue.
- $void\ queueEnqueue(PQueue\ queue, void\ *data)$ – Adds an item to the queue.[311]
- $void\ *\ queueDequeue(PQueue\ queue)$ – Removes an item from the queue and returns it. [312]
- $int\ queueIsEmpty(PQueue\ queue)$ – Checks if the queue is empty.[313]

The Queue struct[314] is defined as follows:

- $PQueueNode\ head$ – A pointer to the head of the queue. The head is the first item that was added to the queue, and this field is used to remove items from the queue efficiently in $O(1)$ time.
- $PQueueNode\ tail$ – A pointer to the tail of the queue. The tail is the last item that was added to the queue, and this field is used to add items to the queue efficiently in $O(1)$ time.

---

[309] The Task struct is specifically designed for this assignment, but it can be used for other purposes as well, with minor changes to the code.
[310] The queue API is generic and can be used to create a queue of any type of data. In this assignment, the queue is used to create a queue of tasks. Please note that the queue API is not type safe, and the user must make sure that the data that is added to the queue is of the correct type.
[311] The item is a generic pointer that can point to any type of data.
[312] The item is a generic pointer that can point to any type of data.
[313] Returns 1 if the queue is empty, 0 otherwise.
[314] Please note that direct access to the queue struct fields is not allowed, and the user must use the Queue API to do any operation on the queue, as the functions that enqueue and dequeue items from the queue are what make the queue thread safe in the first place.

- $unsigned\ int\ size$ – The size of the queue.[315] This field is used to check if the queue is empty, and to get the size of the queue in $O(1)$ time.
- $pthread\_mutex\_t\ lock$ – A mutex lock that is used to lock the queue when it's being used by a thread. This field is used to make the queue thread safe.
- $pthread\_cond\_t\ cond$ – A condition variable that is used to signal threads that are waiting for the queue to be unlocked. This field is used to make the queue a blocking queue.

Each queue node[316] has the following fields:

- $void\ *data$ – The data of the node. This is a generic pointer that can point to any type of data.
- $PQueueNode\ next$ – A pointer to the next node in the queue. This field defines the node structure as a singly linked list.

It is recommended to use the macros defined in the $Queue.h$ file to enqueue and dequeue items from the queue. The macros are defined as follows:

- $ENQUEUE(queue, data)$ – Enqueues an item to the queue. The item is a generic pointer that can point to any type of data.
- $DEQUEUE(queue, type)$ – Dequeues an item from the queue and returns it with the correct type. The type is the type of data that was added to the queue.

<u>Important macros definitions</u>:

```
#define MUTEX_INIT(mutex)          pthread_mutex_init(mutex, NULL)
#define MUTEX_LOCK(mutex)          pthread_mutex_lock(mutex)
#define MUTEX_UNLOCK(mutex)        pthread_mutex_unlock(mutex)
#define MUTEX_DESTROY(mutex)       pthread_mutex_destroy(mutex)
#define COND_INIT(cond)            pthread_cond_init(cond, NULL)
#define COND_WAIT(cond, mutex)     pthread_cond_wait(cond, mutex)
#define COND_SIGNAL(cond)          pthread_cond_signal(cond)
#define COND_DESTROY(cond)         pthread_cond_destroy(cond)
```

Active Object API

The Active Object API supports the following operations:

- $PActiveObject\ CreateActiveObject(PQueueFunc\ func)$ – Creates a new active object, a new queue and starts the active object thread. Returns a pointer to the active object if the operation was successful, or NULL if an error occurred.
- $PQueue\ getQueue(PActiveObject\ activeObject)$ – Returns the queue of the active object, or NULL if an error occurred.
- $void\ stopActiveObject(PActiveObject\ activeObject)$ – Stops the active object thread, destroys the queue, and frees the memory allocated for the active object.
- $void\ *activeObjectRunFunction(void\ *activeObject)$ – The function that is executed by the active object thread. This function is responsible for dequeuing tasks from the queue and executing them, not to be called by the user directly.

[315] Number of items in the queue.
[316] Please note that direct access to the queue struct fields is not allowed, and the user must use the Queue API to do any operation on the queue, as the functions that enqueue and dequeue items from the queue are what make the queue thread safe in the first place.

The Active Object struct has the following fields:

- $pthread\_t\ thread$ – The thread of the active object. Users can use this field to join the thread, but not to cancel it. To cancel the thread, use the $stopActiveObject$ function.
- $PQueue\ queue$ – The queue of the active object, which holds the tasks that the active object will execute.
- $PQueueFunc\ func$ – A function pointer to the handler function of the active object, which will be executed by the active object thread anytime there is a task in the queue.

Each active object receives a function pointer to a function that will be executed by the active object thread. The signature of the handler function is:

$$int\ handler\_function(void\ *task)$$

Where $task$ is the data that was dequeued from the queue. The handler function must return 1 if the active object should continue running, or 0 if the active object should stop running. The handler function is responsible for freeing the memory allocated for the task, and for handling any errors that may occur during the execution of the task.

Actual implementation:[317]

***Task.c*:**

```
#include "Task.h"
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <math.h>
int check_prime(unsigned int num) {
	if (num <= 2 || num % 2 == 0) return 0;
	unsigned int root = (unsigned int)sqrt(num);
	for (unsigned int i = 3; i <= root; i += 2)
		if (num % i == 0) return 0;
	return 1;
}
PTask createTask(unsigned int num_of_tasks, unsigned int _data) {
	PTask task = (PTask)malloc(sizeof(Task));
	task->num_of_tasks = num_of_tasks;
	task->_data = _data;
	return task;
}
void destroyTask(PTask task) {
	free(task);
}
```

[317] Error checking and most debugging functions were removed to simplify the code itself. Full code is available in GitHub.

***Queue.c*:**

```
#include "Queue.h"
#include <stdio.h>
#include <stdlib.h>
PQueue queueCreate() {
	PQueue queue = (PQueue)malloc(sizeof(Queue));
	queue->head = NULL;
	queue->tail = NULL;
	queue->size = 0;
	MUTEX_INIT(&queue->lock);
	COND_INIT(&queue->cond);
	return queue;
}
void queueDestroy(PQueue queue) {
	MUTEX_LOCK(&queue->lock);
	PQueueNode node = queue->head;
	while (node != NULL)
	{
		PQueueNode next = node->next;
		free(node->data);
		free(node);
		node = next;
	}
	MUTEX_UNLOCK(&queue->lock);
	COND_DESTROY(&queue->cond);
	MUTEX_DESTROY(&queue->lock);
	free(queue);
}
void queueEnqueue(PQueue queue, void *data) {
	PQueueNode node = (PQueueNode)malloc(sizeof(QueueNode));
	node->data = data;
	node->next = NULL;
	MUTEX_LOCK(&queue->lock);
	if (queue->head == NULL) {
		queue->head = node;
		queue->tail = node;
		COND_SIGNAL(&queue->cond);
	} else {
		queue->tail->next = node;
		queue->tail = node;
	}
	queue->size++;
	MUTEX_UNLOCK(&queue->lock);
}
void *queueDequeue(PQueue queue) {
	MUTEX_LOCK(&queue->lock);
	while (queue->head == NULL) COND_WAIT(&queue->cond, &queue->lock);
	PQueueNode node = queue->head;
	void *data = node->data;
	queue->head = node->next;
	if (queue->head == NULL) queue->tail = NULL;
	free(node);
	queue->size--;
	MUTEX_UNLOCK(&queue->lock);
	return data;
}
int queueIsEmpty(PQueue queue) {
	MUTEX_LOCK(&queue->lock);
	int isEmpty = (queue->size == 0);
	MUTEX_UNLOCK(&queue->lock);
	return isEmpty;
}
```

ActiveObject.c:

```
#include "ActiveObject.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
PActiveObject CreateActiveObject(PQueueFunc func) {
	static unsigned int id = 0;
	PActiveObject activeObject = (PActiveObject)malloc(sizeof(ActiveObject));
	activeObject->queue = queueCreate();
	activeObject->func = func;
	activeObject->id = id++;
	int ret = pthread_create(&activeObject->thread, NULL, activeObjectRunFunction, activeObject);
	return activeObject;
}
PQueue getQueue(PActiveObject activeObject) {
	return activeObject->queue;
}
void stopActiveObject(PActiveObject activeObject) {
	unsigned int id = activeObject->id;
	pthread_cancel(activeObject->thread);
	activeObject->func = NULL;
	pthread_join(activeObject->thread, NULL);
	queueDestroy(activeObject->queue);
	free(activeObject);
}

void *activeObjectRunFunction(void *activeObject) {
	PActiveObject ao = (PActiveObject)activeObject;
	PQueue queue = ao->queue;
	void *task = NULL;
	while (ao->func && ((task = DEQUEUE(queue, void *))))
		if (ao->func(task) == 0) break;
	return activeObject;
}
```

Tasks.c:

```
#include "Tasks.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
extern PActiveObject *ActiveObjects_Array;
int ActiveObjectTask1(void *task) {
	PTask task_init = (PTask)task;
	unsigned int n = task_init->num_of_tasks, seed = task_init->_data;
	srand(seed != 0 ? seed : time(NULL));
	for (unsigned int i = 0; i < n; i++) {
		unsigned int num = (rand() % 900000) + 100000;
		PTask task_data = createTask(n, num);
		ENQUEUE(getQueue(*(ActiveObjects_Array + 1)), task_data);
		usleep(1000);
	}
	return 0;
}
int ActiveObjectTask2(void *task) {
	static unsigned int count = 0;
	PTask task_data = (PTask)task;
	unsigned int iterations = task_data->num_of_tasks, num = task_data->_data;
	fprintf(stdout, "%u\n%s\n", num, check_prime(num) ? "true" : "false");
	destroyTask(task_data);
	task_data = createTask(iterations, (num + 11));
	ENQUEUE(getQueue(*(ActiveObjects_Array + 2)), task_data);
	return (iterations <= ++count) ? 0 : 1;
}

int ActiveObjectTask3(void *task) {
	static unsigned int count = 0;
	PTask task_data = (PTask)task;
	unsigned int iterations = task_data->num_of_tasks, num = task_data->_data;
	fprintf(stdout, "%u\n%s\n", num, check_prime(num) ? "true" : "false");
	destroyTask(task_data);
	task_data = createTask(iterations, (num - 13));
	ENQUEUE(getQueue(*(ActiveObjects_Array + 3)), task_data);
	return (iterations <= ++count) ? 0 : 1;
}

int ActiveObjectTask4(void *task) {
	static unsigned int count = 0;
	PTask task_data = (PTask)task;
	unsigned int iterations = task_data->num_of_tasks, num = task_data->_data;
	fprintf(stdout, "%u\n%s\n%u\n", num, check_prime(num) ? "true" : "false", (num + 2));
	destroyTask(task_data);
	return (iterations <= ++count) ? 0 : 1;
}
```

Main.c:

```
#include "Tasks.h"
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
PActiveObject *ActiveObjects_Array = NULL;
PTask task_init = NULL;
int main(int argc, char **args) {
	PQueueFunc Functions_Array[ACTIVE_OBJECTS_NUM] = { ActiveObjectTask1, ActiveObjectTask2, ActiveObjectTask3, ActiveObjectTask4 };
	unsigned int n = 0, seed = 0;
	switch(argc)
	{
		case 1:
			fprintf(stderr, "Usage: %s <n> [<seed>]\n", *args);
			return 1;
		case 2:
			if (atoi(*(args + 1)) < 0) {
				fprintf(stderr, "Error: n must be a positive integer.\n");
				return 1;
			}
			n = atoi(*(args + 1));
			break;
		case 3:
			if (atoi(*(args + 1)) <= 0) {
				fprintf(stderr, "Error: n must be a positive integer.\n");
				return 1;
			} else if (atoi(*(args + 2)) <= 0) {
				fprintf(stderr, "Error: seed must be a positive integer.\n");
				return 1;
			}
			n = atoi(*(args + 1));
			seed = atoi(*(args + 2));
			break;
		default:
			fprintf(stderr, "Usage: %s <n> [<seed>]\n", *args);
			return 1;
	}
	ActiveObjects_Array = (PActiveObject *) malloc(sizeof(PActiveObject) * ACTIVE_OBJECTS_NUM);
	task_init = createTask(n, seed);
	for (int i = 0; i < ACTIVE_OBJECTS_NUM; i++)
		*(ActiveObjects_Array + i) = CreateActiveObject(*(Functions_Array + i));
	ENQUEUE(getQueue(*(ActiveObjects_Array)), task_init);
	for (int i = 0; i < ACTIVE_OBJECTS_NUM; i++)
		pthread_join((*(ActiveObjects_Array + i))->thread, NULL);
	for (int i = 0; i < ACTIVE_OBJECTS_NUM; i++)
		stopActiveObject(*(ActiveObjects_Array + i));
	destroyTask(task_init);
	free(ActiveObjects_Array);
	return 0;
}
```

Assignment 6

Context

To solve this assignment, it's recommended to read the Linux Kernel Module Programming Guide. The assignment is based on chapters 1 through 9 and 12 in the book. The purpose of this assignment is to verify that you know the kernel API as much as possible. In this assignment we'll implement a shared memory space that's encrypted, using character device. The encryption function is XOR, and it'll encrypt each 4 bytes of data using a key that's provided using via $ioctl(2)$ system call.

The task is as followed:

- You'll need to create a character device that represents the encrypted memory.[318]
- You'll need to provide support to $ioctl(2)$ system calls, to allow the user to set or change the key.[319] In case of changing the key, the device needs to decrypt the data and encrypt it back using the new key.
- The initial key must be obtained using a parameter.[320]
- You must add support for storing the data using sequence.[321]
- Everything that's written to the character device,[322] must be encrypted with XOR and saved to the memory of the device.[323]
- Everything that was read, we can directly copy.[324]
- You can add support to multiple writes using pages in sequence.
- You need to create a file in $/sys$ that will represent the number of bytes that are currently stored in the device.[325]
- You don't need to support the $lseek(2)$ system call.
- If the sequence is ended, meaning that we reached EOF, you'll restart the sequence from the beginning.
- You'll need to support locking, meaning you must make the device thread safe.[326]
- The character device should be created automatically,[327] and also supports creating it with the $mknod(2)$ system call.
- You don't need to support two-character devices with two different minor numbers.
- It’s necessary to issue a significant log describing the actions taken that can be read with $dmesg(1)$ cmd tool command.

[318] See chapter 6 in the guide.
[319] See chapter 9 in the guide.
[320] See chapter 4 in the guide, program $hello - 5.ko$.
[321] See chapter 7 in the guide.
[322] Meaning, is written with $write(2)$ system call.
[323] See chapter 6 in the guide.
[324] The written data is already encrypted.)(See chapter 6 in the guide.
[325] See chapter 8 in the guide.
[326] See chapter 12 in the guide.
[327] See chapter 6 in the guide.

Solution

chardev.c:

```
#include <linux/atomic.h>
#include <linux/cdev.h>
#include <linux/delay.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/kernel.h> /* for sprintf() */
#include <linux/module.h>
#include <linux/printk.h>
#include <linux/types.h>
#include <linux/uaccess.h> /* for get_user and put_user */
#include <asm/errno.h>
#include <linux/mutex.h>
#include <linux/kobject.h>
#include <linux/string.h>
#include <linux/sysfs.h>
#include <linux/proc_fs.h> /* Necessary because we use proc fs */
#include <linux/seq_file.h> /* for seq_file */
#include <linux/version.h>
#define PROC_NAME "iter"
static int device_open(struct inode *, struct file *);  //open
static int device_release(struct inode *, struct file *); //release
static ssize_t device_read(struct file *, char __user *, size_t, loff_t *); //chardev read
static ssize_t device_write(struct file *, const char __user *, size_t, loff_t *); //chardev write
static long int device_ioctl(struct file *,unsigned int ioctl_num , unsigned long ioctl_param); //IO control
static ssize_t bytes_registered_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf) ; // \sys\kernel\chardev\bytes_registered
static void *my_seq_start(struct seq_file *s, loff_t *pos) ; //seqfile start()
static void *my_seq_next(struct seq_file *s, void *v, loff_t *pos); //seqfile next()
static void my_seq_stop(struct seq_file *s, void *v); //seqfile stop();
static int my_seq_show(struct seq_file *s, void *v);//seqfile show()
#define SUCCESS 0
#define DEVICE_NAME "chardev"
#define BUFFER 64 // BUFFER SIZE
#define IOCTL_CHANGEKEY _IOW('n' ,1,int) //ioctl cmd for case we get new key
static int major; // major  number assigned to our device
volatile static int is_open=0; // if we opened device
static int write_offset=0; //where to write on the next write call
static int key_pos=0; //for our encryption procedure
static char message[BUFFER]; //where our data is located
static unsigned int buffer_size=0; // length of message how many bytes writed in BUFFER
static int key=0;// key to encrypt data message[0-4] xor key because key is 4 bytes
module_param(key, int , 0000); //decleration so we can assign on insmod chardev.ko key=(some value)
MODULE_PARM_DESC(key, "An integer");
static struct class *cls;
static DEFINE_MUTEX(mymutex);
static int bytes_registered=0;
static struct kobject *mykobject;
static struct kobj_attribute bytes_registered_attr = __ATTR(bytes_registered, 0660, bytes_registered_show, NULL);
static struct file_operations chardev_fops = {
    .owner = THIS_MODULE,
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release,
    .unlocked_ioctl = device_ioctl,

};
static struct seq_operations my_seq_ops = {
    .start = my_seq_start,
    .next = my_seq_next,
    .stop = my_seq_stop,
    .show = my_seq_show,
};
static int __init chardev_init(void){
    struct proc_dir_entry *entry;
    printk(KERN_INFO "(chardev init)\n");
    memset(message,0,BUFFER);
    mutex_init(&mymutex);
    major=register_chrdev(0,DEVICE_NAME,&chardev_fops);
    if(major<0){
        pr_alert("Register major failed\n");
        return major;
    }
    pr_info("registered major %d number\n",major);
    buffer_size=strlen(message);
    mykobject = kobject_create_and_add("chardev",kernel_kobj);
    if(!mykobject){
        return -ENOMEM;
    }
    if(sysfs_create_file(mykobject,&bytes_registered_attr.attr)){
        pr_info("failed to create the bytes_regitered file "
                "in /sys/kernel/chardev\n");
    }
    entry = proc_create_seq(PROC_NAME,0,NULL,&my_seq_ops);
    if (entry == NULL) {
        remove_proc_entry(PROC_NAME, NULL);
        pr_debug("Error: Could not initialize /proc/%s\n", PROC_NAME);
        return -ENOMEM;
    }
    cls = class_create(THIS_MODULE, DEVICE_NAME);
    device_create(cls, NULL, MKDEV(major, 0), NULL, DEVICE_NAME);
    printk(KERN_INFO "key = %d\n",key);
    pr_info("device created on /dev/%s \n",DEVICE_NAME);
    return SUCCESS;
}
```

```
static void __exit chardev_exit(void){
    kobject_put(mykobject);
    device_destroy(cls, MKDEV(major, 0));
    class_destroy(cls);
    remove_proc_entry(PROC_NAME, NULL);
    pr_debug("/proc/%s removed\n", PROC_NAME);
    unregister_chrdev(major, DEVICE_NAME);
}
static int device_open(struct inode * iNode, struct file * myFile){
    int ret=0;
    ret = mutex_trylock(&mymutex);
    if(ret!=0) printk(KERN_INFO "mutex is locked\n");
    if(is_open==1){
        printk(KERN_INFO "already open\n");
        return -EBUSY;
    }
    is_open =1;
    try_module_get(THIS_MODULE);
    return SUCCESS;
}
static int device_release(struct inode * iNode, struct file * myFile){
    if(is_open==0){
        printk(KERN_INFO "ERROR - device  wasnt opened\n");
        return -EBUSY;
    }
    is_open =0;
    module_put(THIS_MODULE);
    mutex_unlock(&mymutex);
    if (mutex_is_locked(&mymutex) == 0) printk("Tmutex unlock!\n");
    return SUCCESS;
}
static ssize_t device_read(struct file * myFile, char __user * buffer, size_t sizeT, loff_t * offset){
    int bytes_read=0;//how many bytes we read
    int read_offset=0;
    unsigned char* bytes_key =(unsigned char*)&key;// casting our key to 4 bytes each byte will be represented as char
    size_t keylen=strlen(bytes_key);
    if(offset ==NULL) return -1;
    while((bytes_read<BUFFER) && (*offset < buffer_size)){
        message[read_offset]=message[read_offset]^bytes_key[key_pos];//decrypt
        put_user(message[read_offset],&buffer[*offset]);//printing on user output
        message[read_offset]=message[read_offset]^bytes_key[key_pos];//encrypt
        bytes_read++;
        read_offset++;
        *offset=*offset+1;
        if(read_offset>=BUFFER-1) read_offset=0;
        mutex_unlock(&mymutex);
    }
    printk(KERN_INFO "%s\n",message);
    printk(KERN_INFO "device_read\n");
    return bytes_read;
}
static ssize_t device_write(struct file *myFile, const char __user * buffer, size_t sizeT, loff_t * offset){
    int bytes_write=0;
    unsigned char* bytes_key =(unsigned char*)&key;
    size_t keylen=strlen(bytes_key);
    if(offset==NULL){
        return -EINVAL;
    }
    while(bytes_write< sizeT){
        get_user(message[write_offset], &buffer[bytes_write]);
        key_pos=write_offset % keylen;
        message[write_offset]=message[write_offset]^bytes_key[key_pos];//same as read encrypted here
        bytes_write++;
        write_offset++;
        if(write_offset>=BUFFER-1) write_offset=0;
    }
    buffer_size=strlen(message);
    bytes_registered=buffer_size;
    printk(KERN_INFO "device_write\n");
    printk(KERN_INFO "(%s)",message);
    return bytes_write;
}
```

```
static long int device_ioctl(struct file *,unsigned int ioctl_num , unsigned long ioctl_param){
    int newkey=0;//for the new key we get from mioctl
    int keylen=0;//len of key 4bytes
    int idx=0,keypos=0;//same for read
    unsigned char* bytes_key =NULL;
    switch(ioctl_num){
        case IOCTL_CHANGEKEY:
            if(copy_from_user(&newkey, (int __user*)ioctl_param,sizeof(int))) return -EFAULT;
            if(key==newkey) break;
            printk(KERN_INFO "new key %d\n",newkey);
            printk(KERN_INFO "old key encryption (%s)",message);
            //decrypt using old key
            bytes_key = (unsigned char*)&key;
            keylen= strlen(bytes_key);
            for(idx=0; idx<buffer_size;idx++){
                keypos= idx % keylen;
                message[idx]=message[idx]^bytes_key[keypos];
            }
            printk(KERN_INFO "decrypted (%s)",message);
            key=newkey;
            bytes_key = (unsigned char*)&key;
            keylen= strlen(bytes_key);
            for(idx=0; idx<buffer_size;idx++){
                keypos= idx % keylen;
                message[idx]=message[idx]^bytes_key[keypos];
            }
            printk(KERN_INFO "newkey encryption (%s)",message);
            break;
        default: return -EINVAL;
    }
    return SUCCESS;
}
static ssize_t bytes_registered_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf) {
    return sprintf(buf,"bytes written %d\n",bytes_registered);
}
static size_t * pcounter=NULL;
static void *my_seq_start(struct seq_file *s, loff_t *pos){
    static size_t counter = 0;
    pcounter=&counter;
    if (*pos == 0) return &counter;
    *pos = 0;
    return NULL;
}
static void *my_seq_next(struct seq_file *s, void *v, loff_t *pos){
    size_t *tmp_v = (size_t *)v;
    (*tmp_v)++;
    (*pos)++;
    return NULL;
}
static void my_seq_stop(struct seq_file *s, void *v){
    if(!(*pcounter<buffer_size-1)) *pcounter=0;
}
static int my_seq_show(struct seq_file *s, void *v){
    size_t idx=*(size_t*)v;
    seq_printf(s, "%c\n", message[idx]);
    return 0;
}
module_init(chardev_init);
module_exit(chardev_exit);
MODULE_LICENSE("GPL");
```

mioctl.c:[328]

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#define IOCTL_CHANGEKEY _IOW('n' ,1,int)
#define MAJOR_NUM 509
int main(int argc,char *argv[]) {
    if(argc<2){
        printf("Usage: %s <new_key>\n", argv[0]);
        return -1;
    }
    int fd = open("/dev/chardev", O_RDWR);
    if (fd < 0) {
        perror("Failed to open device file");
        return -1;
    }
    int newKey = atoi(argv[1]);
    if (ioctl(fd,  IOCTL_CHANGEKEY, &newKey) < 0) {
        perror("ioctl failed");
        close(fd);
        return -1;
    }
    printf("IOCTL command executed successfully\n");
    close(fd);
    return 0;
}
```

[328] This is a user-space demo program to change the encryption key using the $ioctl(2)$ system call.

Assignment 4 Bonus

Context

Part A – Generic Proactor Library (1 pts):

Read in Wikipedia or on Pattern-Oriented Software Architecture (POSA) about the Proactor design pattern, which is the opposite of the Reactor design pattern.

Implements Beej's chat ($poll(2)$ or $select(2)$ version to your choice), using the Proactor design pattern.

The Proactor will open a thread that will send the message to all other clients that use the chat, that will run as soon as a message is received by the server. After the thread is finished with sending the message to all the users in the chat, the thread will die.

Functions to implement:

- $typedef\ int\ (handler\_t\ *)(int)$ – A function that handles a "hot" file descriptor, receives a file descriptor, and returns 0 when succeeded.
- $void\ *\ createProactor(void\ *\ args)$ – A function that allocates and creates a Proactor. You decide what will be the structure of the Proactor object, and if the function will receive any arguments.
- $int\ runProactor(void\ *\ this)$ – Will run the Proactor's thread. Returns 0 on success.
- $int\ cancelProactor(void\ *\ this)$ – Stops Proactor's thread. Returns 0 on success.
- $int\ addFD2Proactor(void\ *\ this, int\ fd, handler\_t\ handler)$ – Adds a file descriptor and a handler function to the Proactor. Returns 0 on success.
- $int\ removeHandler(void\ *\ this, int\ fd)$ – Removes a file descriptor from the Proactor. Returns 0 on success.

Part B – Using the Proactor (1 pts):

Build a Makefile which compiles the Proactor's library and a program that's an implementation of the Beej's chat using the Proactor library.

Solution

Proactor Library:

The Proactor library supports the following functions:

- $void\ *createProactor()$ – Create a proactor object - a linked list of file descriptors and their handlers.
- $int\ runProactor(void\ *this)$ – Start executing the proactor, in a new thread.
- $int\ cancelProactor(void\ *this)$ – Gracefully stop the proactor - stop the proactor thread.
- $int\ addFD2Proactor(void\ *this, int\ fd, handler\_t\ handler)$ – Add a file descriptor to the proactor.[329]
- $int\ removeHandler(void\ *this, int\ fd)$ – Remove a file descriptor from the proactor.[330]
- $int\ destroyProactor(void\ *this)$ – Destroy the proactor - stop the proactor thread and free all the memory it allocated.

The signature of the handler function for proactors is: $int\ handler\_t(int\ fd)$. The function should return 0 on success, or 1 on failure.

[329] You should never add the listening socket to the proactor, only the client sockets. Trying to add the listening socket to the proactor will result in a crash, as sending messages to the listening socket is not supported.

[330] The proactor will automatically remove the file descriptor from the proactor when it encounters an error, so you don't need to remove it yourself.

Implementation:[331]

```
#include "proactor.h"
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
void *proactorRunFunction(void *args) {
	PProactor proactor = (PProactor)args;
	PProactorNode curr = proactor->head;
	while (curr != NULL && proactor->isRunning) {
		if (curr->hdlr.handler != NULL) {
			curr->hdlr.handler(curr->fd);
		}
		curr = curr->next;
	}
	proactor->isRunning = false;
	pthread_exit(proactor);
	return NULL;
}
void *createProactor() {
	PProactor proactor = (PProactor) malloc(sizeof(Proactor));
	proactor->thread = 0;
	proactor->head = NULL;
	proactor->isRunning = false;
	proactor->size = 0;
	return proactor;
}
int runProactor(void *this) {
	PProactor proactor = (PProactor)this;
	proactor->isRunning = true;
	pthread_create(&proactor->thread, NULL, proactorRunFunction, proactor);
	return 0;
}
int cancelProactor(void *this) {
	PProactor proactor = (PProactor)this;
	proactor->isRunning = false;
	pthread_cancel(proactor->thread);
	pthread_join(proactor->thread, NULL);
	return 0;
}
int addFD2Proactor(void *this, int fd, handler_t handler) {
	PProactor proactor = (PProactor)this;
	PProactorNode node = (PProactorNode) malloc(sizeof(ProactorNode));
	node->fd = fd;
	node->hdlr.handler = handler;
	node->next = NULL;
	if (proactor->head == NULL) {
		proactor->head = node;
	} else {
		PProactorNode curr = proactor->head;
		while (curr->next != NULL)
			curr = curr->next;
		curr->next = node;
	}
	proactor->size++;
	return 0;
}
int removeHandler(void *this, int fd) {
	PProactor proactor = (PProactor)this;
	PProactorNode curr = proactor->head;
	if (curr->fd == fd) {
		proactor->head = curr->next;
		free(curr);
		proactor->size--;
		return 0;
	}
	while (curr->next != NULL) {
		if (curr->next->fd == fd) {
			PProactorNode tmp = curr->next;
			curr->next = curr->next->next;
			free(tmp);
			proactor->size--;
			return 0;
		}
		curr = curr->next;
	}
	return 1;
}
int destroyProactor(void *this) {
	PProactor proactor = (PProactor)this;
	if (proactor->isRunning)
		cancelProactor(proactor);
	if (proactor->head != NULL) {
		PProactorNode curr = proactor->head;
		while (curr != NULL) {
			PProactorNode tmp = curr;
			curr = curr->next;
			free(tmp);
		}
	}
	free(proactor);
	return 0;
}
```

[331] For simplicity, no error checking is included. Full code is avaliable on GitHub.

Proactor Server Demo implementation:[332]

```
#include "reactor.h"
#include "proactor.h"
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <errno.h>
#include <signal.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
void *reactor = NULL;
void *proactor = NULL;
char *message = "This is a message from the server! "
                "A client has sent a message to the server, "
                "and the server is sending this message back "
                "to the client via the proactor.\n";
int main(void) {
        struct sockaddr_in server_addr = {
                .sin_family = AF_INET,
                .sin_port = htons(SERVER_PORT),
                .sin_addr.s_addr = INADDR_ANY
        };
        int server_fd = -1, reuse = 1;
        signal(SIGINT, signal_handler);
        socket(AF_INET, SOCK_STREAM, 0);
        setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(int));
        bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
        listen(server_fd, MAX_QUEUE);
        reactor = createReactor();
        proactor = createProactor();
        addFd(reactor, server_fd, server_handler);
        fprintf(stdout, "Server socket added to reactor successfully.\n", );
        startReactor(reactor);
        WaitFor(reactor);
        return EXIT_SUCCESS;
}
void *client_handler(int fd, void *react) {
        char *buf = (char *)calloc(MAX_BUFFER, sizeof(char));
        int bytes_read = recv(fd, buf, MAX_BUFFER, 0);
        if (bytes_read <= 0) {
                fprintf(stdout, "%s Client %d disconnected.\n", fd);
                removeHandler(proactor, fd);
                free(buf);
                close(fd);
                return NULL;
        }
        if (bytes_read < MAX_BUFFER)
                *(buf + bytes_read) = '\0';
        else
                *(buf + MAX_BUFFER - 1) = '\0';
        fprintf(stdout, "%s Client %d: %s\n", fd, buf);
        free(buf);
        runProactor(proactor);
        pthread_join(((PProactor)proactor)->thread, NULL);
        return react;
}
void *server_handler(int fd, void *react) {
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);
        reactor_t_ptr reactor = (reactor_t_ptr)react;
        int client_fd = accept(fd, (struct sockaddr *)&client_addr, &client_len);
        fprintf(stdout, "Client %s:%d connected, Reference ID: %d\n", inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port), client_fd);
        addFd(reactor, client_fd, client_handler);
        addFD2Proactor(proactor, client_fd, fds_handler);
        return react;
}
int fds_handler(int fd) {
        send(fd, message, strlen(message), 0);
        return 0;
}
```

[332] For simplicity, no error checking is included. Full code is avaliable on GitHub. The Proactor server demo is using the Reactor library from the solution of assignment 4.

Assignment 5 Bonus

Context

This assignment uses pipeline, external libraries and implements a simple mail server.

Part A:

- Read the functionally of $uuencode(1)$ and $uudecode(1)$. Those functions allow to encode binary files to base64, meaning using 64 ASCII printable characters. In this way, we can encode 4 characters of base64,[333] to 3 characters that use all the 8 bits.[334]
- Read about compression using the Bzip2 algorithm and library. You can read the manual in the **following link**.
- Read about OpenSSL's encryption function that use the AES algorithm:

$$AES\_set\_encrypt\_key(key, 128, \&enc\_key);$$

$$AES\_encrypt(text, enc\_out, \&enc\_key);$$

$$AES\_set\_decrypt\_key(key, 128, \&dec\_key);$$

$$AES\_decrypt(enc\_out, dec\_out, \&dec\_key);$$

We'll now implement a client and a server mail that uses the VSMTP (Very Simple Mail Transport Protocol) protocol, which is a custom-made protocol that's based on SMTP.

In the VSMTP protocol, there is only one mail server per client, and there is only one client per mail server. A client's address is the server's IP address. There is no authentication. Anyone that connects to the server and claims that he's the client or that he has an email for the only client in the server, the server allows him to read mails or to send to his client a mail.

Mail can contain text and 0 or more attachments. The mail also contains an address field and a subject field.

To send the email we connect directly to the remote server[335] and sends him:

- Our address.[336]
- The subject of the mail.[337]
- The mail content.[338]
- The mail's attachments.[339]

The address passes as it is in the $accept(2)$ system call, no further process needed. The subject, content and the attachments are encoded, compressed, and encrypted before they are sent to the server.

---

[333] Which are 24 bits in total.
[334] Which are also 24 bits in total.
[335] In SMTP we connect to our local SMTP server.
[336] From whom the mail was sent.
[337] Could be empty.
[338] Could be empty.
[339] 0 or more.

Assume that the mail server needs to support IPv4 only and all the VSMTP server are running on known, fixed port.[340]

To receive mails, we need to connect to the local server, and it sends us the list of all the emails that are stored in the server, in the form of address and a subject. We can choose if we want to download a message and it will be downloaded and shown to us. Moreover, all its attachments will be downloaded and will be saved to a special folder of your choice.

The client can also request to download all the emails that are stored in the server in a single command.

The VSMTP protocol always sends the messages after they were encoded in base64, all the attachments are attached in a linked list, the content as a whole is compressed and encrypted. Only the clients themselves can open or compress the message, as the server always saves the messages when they are encoded, compressed, and encrypted.

The messages and the files will be shown to us as decoded, uncompressed, and decrypted, and we need to encode, compress, and encrypt them before sending them.

Important notes:

- You need to implement a pipeline that does compression and decryption. The pipeline will include at least one Active Object for each task,[341] and can also include extra Active Object that can, for example, save the files and showing the message to the client.
- You don't need to support a standard, clients, or servers of other students, but your server must work with your client.[342]
- You must supply a Makefile that builds the client and the server.
- You must use the pipeline library that you implemented in assignment 5.
- OpenSSL's library also supports base64 encoding and decoding. Don't use it, use the $uuencode(1)$ and the $uudecode(1)$ functions.

[340] Meaning the IP field itself determines the address of the sender.
[341] Decrypt, Decompress and decode.
[342] You can configure this communication how you want.

Solution

Task API

The Task API supports the following operations:

- $PTask\ createTask(void\ *data, u\_int32\_t\ size)$ – Creates a new task with the given data ($data$ and $size$), allocates memory for it and returns a pointer to the task.
- $void\ taskDestroy(PTask\ task)$ – Destroys a task - frees the memory allocated for the task.

The Task struct has the following fields:

- $void\ *\_data$ – The data to pass to the active object. The data has generic purpose and can be used for any purpose by the active object.
- $u\_int32\_t\ \_data\_size$ – The size of the data to pass to the active object. This indicates the size of the data to pass to the active object and is used for memory allocation purposes.

Linked List API

The Linked List API supports the following operations:

- $PLinkedList\ createLinkedList()$ – Creates a new linked list and returns a pointer to the list.
- $int\ addNode(PLinkedList\ list, void\ *data)$ – Adds a new node to the end of the list.
- $int\ removeNode(PLinkedList\ list, void\ *data)$ – Removes a node from the list.
- $void\ *getHead(PLinkedList\ list)$ – Returns the data stored in the head of the list.
- $void\ *getTail(PLinkedList\ list)$ – Returns the data stored in the tail of the list.
- $int\ destroyLinkedList(PLinkedList\ list)$ – Destroys a linked list.

The LinkedList struct has the following fields:

- $PNode\ head$ – A pointer to the head of the list. The head pointer is NULL if the list is empty. The head pointer should not be changed directly.
- $PNode\ tail$ – A pointer to the tail of the list. The tail pointer is NULL if the list is empty. The tail pointer should not be changed directly.

The Node struct has the following fields:

- $void\ *data$ – A pointer to the data stored in the node. The data pointer is void, so it can be used to store any type of data. The data pointer cannot be NULL.
- $PNode\ next$ – A pointer to the next node in the list. The next pointer is NULL if the node is the last node in the list. Users should not change the next pointer directly.

Encoding API

The Encoding API supports the following operations:[343]

- $int\ UUEncode(const\ char\ *data, const\ uint32\_t\ data\_size, char\ **encoded\_data, uint32\_t\ *encoded\_data\_size)$ – Encodes the data using the Unencode algorithm. Returns 1 if the operation was successful, or 0 if an error occurred.
- $int\ UUDecode(const\ char\ *encoded\_data, const\ uint32\_t\ encoded\_data\_size, char\ **data, uint32\_t\ *data\_size)$ – Decodes the data using the UUencode algorithm. Returns 1 if the operation was successful, or 0 if an error occurred.

The Encoding API uses the Unencode algorithm, which is an open-source algorithm for encoding and decoding data.

Compression API

The Compression API supports the following operations:

- $int\ Bzip2\_Compress(void\ *\ data, uint32\_t\ size, void\ **\ compressed\_data, uint32\_t\ *\ compressed\_size)$ – Compresses the data using the Bzip2 algorithm. Returns 1 if the operation was successful, or 0 if an error occurred.
- $int\ Bzip2\_Decompress(void\ *\ compressed\_data, uint32\_t\ compressed\_size, void\ **\ data, uint32\_t\ *\ size)$ – Decompresses the data using the Bzip2 algorithm. Returns 1 if the operation was successful, or 0 if an error occurred.

The API also have some settings that can be changed in the $Compression.h$ file:

- $BZIP2\_BLOCK\_SIZE$ – Specifies the block size to be used for compression. It should be a value between 1 and 9 inclusive, and the actual block size used is 100000 x this figure. 9 gives the best compression but takes most memory. 1 gives the worst compression but uses least memory.
- $BZIP2\_VERBOSITY$ – Specifies the verbosity level to be used for compression. It should be a value between 0 and 4 inclusive. 0 gives no output, 4 gives maximum output.
- $BZIP2\_WORK\_FACTOR$ – Specifies the work factor to be used for compression. It should be a value between 0 and 250 inclusive.
- $BZIP2\_DECOMPRESS\_SMALL$ – Specifies whether to use the small memory decompression mode or not. It should be 1 to use the small memory decompression mode, or 0 to use the normal mode.

[343] The algorithm is available in the **following link**.

Encryption API

The Encryption API supports the following operations:

- $int\ AES\_func\_encrypt\_data(uint8\_t\ *\ plaintext, int\ plaintext\_len, const\ uint8\_t\ *key, const\ uint8\_t\ *iv, uint8\_t\ *\ ciphertext)$ – Encrypts the data using the $AES-256-CBC$ algorithm. Returns length of the ciphertext if the operation was successful, or 0 if an error occurred.
- $int\ AES\_func\_decrypt\_data(uint8\_t\ *\ ciphertext, int\ ciphertext\_len, const\ uint8\_t\ *key, const\ uint8\_t\ *\ iv, uint8\_t\ *data)$ – Decrypts the data using the $AES-256-CBC$ algorithm. Returns length of the data if the operation was successful, or 0 if an error occurred.

The key and the initialization vector (IV) that are used for encryption and decryption can be changed in the $Encryption.h$ file:

- EVP_AES_KEY – The key that is used for encryption and decryption. The key is a 256-bit key, and it is represented as a 32-byte array.
- EVP_AES_IV – The initialization vector (IV) that is used for encryption and decryption. The IV is a 128-bit IV, and it is represented as a 16-byte array.

Message API

The Message API supports the following operations:

- $int\ addAttachmentToList(PAttachment\ *head, const\ char\ *name, void\ *\ buffer, const\ uint32\_t\ size)$ – Adds an attachment to the attachments list. Returns 1 if the operation was successful, or 0 if an error occurred.
- $void\ freeAttachmentList(PAttachment\ *head)$ – Frees the attachments list by removing all the attachments from the list and freeing the memory that was allocated for them.
- $uint32\_t\ createMailDataPacket(const\ char\ *from, const\ char\ *\ subject, const\ char\ *body, PAttachment\ attachments, char\ **output)$ – Creates a mail data packet from the given parameters. The whole mail data package is built directly in the output buffer, and the function returns the size of the mail data package.

The message header is built from the following fields:

- $char\ \_mail\_from[USERNAME\_MAX\_LENGTH]$ ($USERNAME\_MAX\_LENGTH$ bytes) – The sender of the mail (Fixed size of $USERNAME\_MAX\_LENGTH$). The sender is used to identify the sender of the mail.
- $char\ \_mail\_subject[MAX\_SUBJECT\_LENGTH]$ ($MAX\_SUBJECT\_LENGTH$ bytes) – The subject of the mail (Fixed size of $MAX\_SUBJECT\_LENGTH$). The subject is used to identify the subject of the mail. It can be empty.
- $uint32\_t\ \_mail\_data\_total\_size$ (4 bytes) – The total size of the mail, including all the fields and the data. The _mail_data_total_size field is calculated by this formula: $\_mail\_data\_total\_size = sizeof(struct\ \_Mail\_Raw\_Packet\_Stripped) + \_mail\_data\_body\_size +$ sum of the sizes of the attachments (if there are any).

- $uint32\_t\ \_mail\_data\_body\_size$ (4 bytes) – The size of the mail body (excluding the attachments). The $\_mail\_data\_body\_size$ field is calculated by this formula:
$$\_mail\_data\_body\_size\ =\ strlen(\_mail\_body)\ +\ 1$$
- $uint32\_t\ \_mail\_attachments\_count$ (4 bytes) – The number of attachments that are attached to the mail.
- $< body\ data >$ (0-? bytes) – The body of the mail. The body is a string, and its size is controlled by the $\_mail\_data\_body\_size$ field. Can be empty.
- $< attachments >$ (0-? bytes) – The attachments of the mail. The attachments are files, and their sizes are controlled by the $\_mail\_attachment\_size$ field. See the attachment header for more details.

The attachment header is built from the following fields:

- $char\ \_mail\_attachment\_name[ATTACH\_FILENAME\_MAX]$ ($ATTACH\_FILENAME\_MAX$ bytes) – The name of the attachment (Fixed size of $ATTACH\_FILENAME\_MAX$). The name is used to identify the attachment.
- $uint32\_t\ \_mail\_attachment\_size$ (4 bytes) – The size of the attachment. The attachment is a file, and its size is controlled by the _mail_attachment_size field.
- $< attachment\ data >$ (0-? bytes) – The data of the attachment. The data is a file, and its size is controlled by the $\_mail\_attachment\_size$ field. Can be empty, but it's illogical to have an attachment with no data.

Implementation:[344]

$Encoding.c$:

```
#include "../include/Encoding.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int UUEncode(const char *data, const uint32_t data_size, char **encoded_data, uint32_t *encoded_data_size) {
	*encoded_data_size = (data_size / 3) * 4;
	if (data_size % 3 != 0)
		*encoded_data_size += 4;
	// Allocate memory for the encoded data
	*encoded_data = (char *)malloc(*encoded_data_size + 1); // +1 for null terminator
	// Perform UUEncode algorithm
	uint32_t i, j;
	for (i = 0, j = 0; i < data_size; i += 3, j += 4) {
		(*encoded_data)[j] = (data[i] & 0xfc) >> 2;
		(*encoded_data)[j + 1] = ((data[i] & 0x03) << 4) | ((data[i + 1] & 0xf0) >> 4);
		(*encoded_data)[j + 2] = ((data[i + 1] & 0x0f) << 2) | ((data[i + 2] & 0xc0) >> 6);
		(*encoded_data)[j + 3] = data[i + 2] & 0x3f;
	}
	// Append newline character to the end of encoded data
	(*encoded_data)[*encoded_data_size] = '\0';
	return 0;
}
int UUDecode(const char *encoded_data, const uint32_t encoded_data_size, char **data, uint32_t *data_size) {
	// Calculate the size of the decoded data
	*data_size = (encoded_data_size / 4) * 3;
	if (encoded_data[encoded_data_size - 1] == '=') {
		*data_size -= 1;
		if (encoded_data[encoded_data_size - 2] == '=')
			*data_size -= 1;
	}
	// Allocate memory for the decoded data
	*data = (char *) malloc(*data_size + 1); // +1 for null terminator
	// Perform UUDecode algorithm
	uint32_t i, j;
	for (i = 0, j = 0; i < encoded_data_size; i += 4, j += 3) {
		(*data)[j] = (encoded_data[i] << 2) | ((encoded_data[i + 1] & 0x30) >> 4);
		(*data)[j + 1] = ((encoded_data[i + 1] & 0x0f) << 4) | ((encoded_data[i + 2] & 0x3c) >> 2);
		(*data)[j + 2] = ((encoded_data[i + 2] & 0x03) << 6) | (encoded_data[i + 3] & 0x3f);
	}
	// Append null terminator to the end of decoded data
	(*data)[*data_size] = '\0';
	return 0;
}
```

[344] To simplify the code, error checking was removed. Full code is available at GitHub.

***Compression.c*:**

```
#include "../include/Compression.h"
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
int Bzip2_Compress(void *data, uint32_t size, void **compressed_data, uint32_t *compressed_size) {
        // Initialize the Bzip2 stream
        bz_stream bzs;
        bzs.bzalloc = NULL;
        bzs.bzfree = NULL;
        bzs.opaque = NULL;
        BZ2_bzCompressInit(&bzs, BZIP2_BLOCK_SIZE, BZIP2_VERBOSITY, BZIP2_WORK_FACTOR);
        // Calculate the size of the compressed data buffer
        uint32_t destSize = size + (size * 0.01) + 600; // BZ2_bzBuffToBuffCompress() recommends adding 1% + 600 bytes
        *compressed_data = malloc(destSize);
        // Perform the compression
        bzs.next_in = (char *)data;
        bzs.avail_in = size;
        bzs.next_out = *compressed_data;
        bzs.avail_out = destSize;
        BZ2_bzCompress(&bzs, BZ_FINISH);
        // Set the compressed size
        *compressed_size = destSize - bzs.avail_out;
        // Clean up the Bzip2 stream
        BZ2_bzCompressEnd(&bzs);
        return 0;
}
int Bzip2_Decompress(void *compressed_data, uint32_t compressed_size, void **data, uint32_t *size) {
        // Initialize the Bzip2 stream
        bz_stream bzs;
        bzs.bzalloc = NULL;
        bzs.bzfree = NULL;
        bzs.opaque = NULL;
        BZ2_bzDecompressInit(&bzs, BZIP2_VERBOSITY, BZIP2_DECOMPRESS_SMALL);
        // Allocate memory for the decompressed data
        *data = malloc(compressed_size * 10); // A rough estimate for the decompressed size
        // Perform the decompression
        bzs.next_in = compressed_data;
        bzs.avail_in = compressed_size;
        bzs.next_out = *data;
        bzs.avail_out = compressed_size * 10;
        BZ2_bzDecompress(&bzs);
        // Set the decompressed size
        *size = compressed_size * 10 - bzs.avail_out;
        // Clean up the Bzip2 stream
        BZ2_bzDecompressEnd(&bzs);
        return 0;
}
```

***Encryption.c*:**

```
#include "../include/Encryption.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
int AES_func_encrypt_data(uint8_t *plaintext, int plaintext_len, const uint8_t *key, const uint8_t *iv, uint8_t *ciphertext) {
        // Initialize the OpenSSL library
        EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
        // Set up the encryption parameters
        EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
        // Perform the encryption
        int len;
        EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len);
        int ciphertext_len = len;
        // Finalize the encryption
        EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);
        ciphertext_len += len;
        // Clean up the OpenSSL context
        EVP_CIPHER_CTX_free(ctx);
        return ciphertext_len;
}
int AES_func_decrypt_data(uint8_t *ciphertext, int ciphertext_len, const uint8_t *key, const uint8_t *iv, uint8_t *plaintext) {
        // Initialize the OpenSSL library
        EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
        // Set up the decryption parameters
        EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
        // Perform the decryption
        int len;
        EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext_len);
        int plaintext_len = len;
        // Finalize the decryption
        (EVP_DecryptFinal_ex(ctx, plaintext + len, &len);
        plaintext_len += len;
        // Clean up the OpenSSL context
        EVP_CIPHER_CTX_free(ctx);
        return plaintext_len;
}
```

***LinkedList.c*:**

```
#include "../include/LinkedList.h"
#include <stdio.h>
#include <stdlib.h>
LinkedList createLinkedList() {
	PLinkedList list = (PLinkedList)malloc(sizeof(LinkedList));
	list->head = NULL;
	list->tail = NULL;
	return list;
}
int addNode(PLinkedList list, void *data) {
	PNode node = (PNode)malloc(sizeof(Node));
	node->data = data;
	node->next = NULL;
	if (list->head == NULL) {
		list->head = node;
		list->tail = node;
	} else {
		list->tail->next = node;
		list->tail = node;
	}
	return 0;
}
int removeNode(PLinkedList list, void *data) {
	PNode node = list->head;
	PNode prev = NULL;
	while (node != NULL) {
		if (node->data == data) {
			if (prev == NULL)
				list->head = node->next;
			else
				prev->next = node->next;
			if (node == list->tail)
				list->tail = prev;
			free(node);
			return 0;
		}
		prev = node;
		node = node->next;
	}
	return 1;
}
void *getHead(PLinkedList list) {
	return list->head->data;
}
void *getTail(PLinkedList list) {
	return list->tail->data;
}
int destroyLinkedList(PLinkedList list) {
	PNode node = list->head;
	PNode next;
	while (node != NULL) {
		next = node->next;
		free(node);
		node = next;
	}
	free(list);
	return 0;
}
```

***Mail.c*:**

```
#include "../include/Mail.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
int addAttachmentToList(PAttachment *head, const char *name, void *buffer, const uint32_t size) {
	if (*head == NULL) {
		*head = (PAttachment)malloc(sizeof(Attachment));
		memcpy((*head)->_attach_name, name, strlen(name) + 1);
		(*head)->_attach_data = (char *)calloc(size, sizeof(char));
		memcpy((*head)->_attach_data, buffer, size);
		(*head)->_attach_data_size = size;
		(*head)->_attach_next = NULL;
	} else {
		PAttachment current = *head;
		while (current->_attach_next != NULL)
			current = current->_attach_next;
		current->_attach_next = (PAttachment)malloc(sizeof(Attachment));
		memcpy(current->_attach_next->_attach_name, name, strlen(name) + 1);
		current->_attach_next->_attach_data = (char *)calloc(size, sizeof(char));
		memcpy(current->_attach_next->_attach_data, buffer, size);
		current->_attach_next->_attach_data_size = size;
		current->_attach_next->_attach_next = NULL;
	}
	return 0;
}
void freeAttachmentList(PAttachment *head) {
	if (*head == NULL) return;
	PAttachment current = *head;
	PAttachment prev = NULL;
	while (current != NULL) {
		prev = current;
		current = current->_attach_next;
		free(prev->_attach_data);
		free(prev);
	}
	*head = NULL;
}
uint32_t createMailDataPacket(const char *from, const char *subject, const char *body, PAttachment attachments, char **output) {
	// Initialize size with the size of the username, subject, total size, body size and attachments number
	uint32_t total_size = USERNAME_MAX_LENGTH + MAX_SUBJECT_LENGTH + (3 * sizeof(uint32_t));
	// Calculate body size
	uint32_t body_size = strlen(body) + 1;
	// Calculate attachments sizes
	uint32_t attachments_size = 0, attachments_count = 0;
	PAttachment current = attachments;
	total_size += body_size;
	while (current != NULL) {
		attachments_size += (ATTACH_FILENAME_MAX + sizeof(uint32_t) + current->_attach_data_size);
		current = current->_attach_next;
		attachments_count++;
	}
	total_size += attachments_size;
	// Allocate memory for the output
	*output = (char *)calloc(total_size, sizeof(char));
	// Copy the username and subject to the output
	memcpy(*output, from, strlen(from) + 1);
	memcpy(*output + USERNAME_MAX_LENGTH, subject, strlen(subject) + 1);
	// Copy the total size, body size and attachments number to the output
	memcpy(*output + USERNAME_MAX_LENGTH + MAX_SUBJECT_LENGTH, &total_size, sizeof(uint32_t));
	memcpy(*output + USERNAME_MAX_LENGTH + MAX_SUBJECT_LENGTH + sizeof(uint32_t), &body_size, sizeof(uint32_t));
	memcpy(*output + USERNAME_MAX_LENGTH + MAX_SUBJECT_LENGTH + (2 * sizeof(uint32_t)), &attachments_count, sizeof(uint32_t));
	// Copy the body to the output
	memcpy(*output + USERNAME_MAX_LENGTH + MAX_SUBJECT_LENGTH + (3 * sizeof(uint32_t)), body, body_size);
	// Copy the attachments to the output
	current = attachments;
	uint32_t offset = USERNAME_MAX_LENGTH + MAX_SUBJECT_LENGTH + (3 * sizeof(uint32_t)) + body_size;
	while (current != NULL) {
		memcpy(*output + offset, current->_attach_name, strlen(current->_attach_name) + 1);
		memcpy(*output + offset + ATTACH_FILENAME_MAX, &(current->_attach_data_size), sizeof(uint32_t));
		memcpy(*output + offset + ATTACH_FILENAME_MAX + sizeof(uint32_t), current->_attach_data, current->_attach_data_size);
		// Recalculate the offset
		offset += (ATTACH_FILENAME_MAX + sizeof(uint32_t) + current->_attach_data_size);
		current = current->_attach_next;
	}
	return total_size;
}
```

Bibliography

Hall, B. (2023). *Beej's Guide to Interprocess Communcation.* From https://beej.us/guide/bgipc/

Hall, B. (2023). *Beej's Guide to Network Programming.* From https://beej.us/guide/bgnet/

IBM. (2021, April 14). *Pthread APIs.* From IBM Documentation: https://www.ibm.com/docs/en/i/7.3?topic=ssw_ibm_i_73/apis/rzah4mst.html

Linux man pages. (n.d.). Retrieved from die.net: https://linux.die.net/man/

Salzman, P. J., Burian, M., Pomerantz, O., Mottram, B., & Huang, J. (2023). *The Linux Kernel Module Programming Guide.* From https://sysprog21.github.io/lkmpg/

Schmidt, D., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects* (Volume 2 edition ed., Vol. 2). Wiley.